IL LINGUAGGIO

Daniel-Jean David

IL LINGUAGGIO ADA

Daniel-Jean David

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

Daniel-Jean David

Daniel-Jean David *insegna informatica di gestione all'Università di Parigi I, Pantheon Sorbonne.*

Insegna anche utilizzazione dei microprocessori a l'E.N.S.A.M. di Parigi.

Gli argomenti di ricerca vanno dalla grafica informatica, alla tecnica di interfaccia dei microprocessori, ai sistemi multiprocessori.

Specialista del microprocessore 6502, ha fatto, a Parigi, numerosi seminari sui microprocessori, il KIM, il SYM e il P.E.T./C.B.M.. È direttore di "La Commode", rivista dedicata ai calcolatori Commodore.

SUPERVISIONE TECNICA: Emi Bennati
TRADUZIONE: Rosalba Canino
GRAFICA E IMPAGINAZIONE: Francesca di Fiore
COPERTINA: Silvana Corbelli
FOTOCOMPOSIZIONE: CorpoNove - Bergamo
STAMPA: A. Matarelli - Milano.

SOMMARIO

APPENDICI

Ritratto di Lady Augusta Ada Byron Contessa di Lovelace, prima "programmatrice" della storia?

PRESENTAZIONE

Nel 1975, il Dipartimento della Difesa degli Stati Uniti, constatando che nei diversi organismi dipendenti erano utilizzati centinaia di linguaggi di programmazione e considerando i costi di conversione richiesti da questa situazione, bandì un concorso per la concezione di un linguaggio che potesse essere impiegato universalmente.

Fu elaborato un quaderno di caratteristiche che specificavano i tipi di dati ammissibili, le strutture di controllo e le qualità di leggibilità e affidabilità richieste. Furono valutati con cura quattro progetti finalizzati ("rosso", "verde", "blu", "giallo").

In definitiva fu il progetto **CII-Honeywell-Bull** *("verde"), elaborato sotto la direzione di Jean* **ICHBIAH**, *ad essere scelto. Questo linguaggio ricevette il nome di ADA in onore alla Contessa Ada Augusta Lovelace (1815-1852), figlia di Lord Byron e musa ispiratrice del geniale Charles Babbage, che concepì il primo calcolatore della storia (quella macchina — a ingranaggi — non funzionò mai in realtà ma incorporava i principali concetti funzionali, per esempio la nozione di programma registrato in memoria, dei calcolatori moderni). A tal titolo, Ada Lovelace è la prima programmatrice della storia.*

Il linguaggio ADA è destinato ad una grande diffusione, su macchine di ogni dimensione: è sostenuto dal Dipartimento della Difesa americano e da altri; un costruttore di microprocessori ha fin d'ora annunciato un insieme di microprocessori che formano una macchina ADA.

ADA suscita già ora un interesse formidabile ed è evidente che, visto il processo di elaborazione e selezione di cui è stato oggetto, deve possedere caratteristiche interessanti.

Per questo ci sembra urgente presentare un opera su ADA, anche se, mentre scriviamo, non esiste un compilatore ADA operativo. Di conseguenza, per quanta cura si sia messa nella verifica dei programmi di esempio che presentiamo, questi non sono garantiti completamente poichè non sono listati usciti direttamente dalla macchina su cui avrebbero dovuto girare.

In particolare, sui punti dove le norme del linguaggio non erano totalmente chiare, noi abbiamo dovuto scegliere una interpretazione che non è necessariamente quella che sarà presa da compilatori diversi.

Questa situazione è diversa da quella degli altri libri Jackson ed è per questo che il libro si intitola "Una introduzione" piuttosto che "Programmare in". Ma ripetiamo che ogni sforzo è stato fatto affinché i programmi presentati siano affidabili e l'Autore si assume la responsabilità di eventuali errori rimasti.

Un punto ancora: l'Autore non fa parte di nessuno dei gruppi che ha contribuito alla definizione di ADA. È dunque un apprezzamento **"dall'esterno"** *che viene presentato per la prima volta e come è nostra abitudine, insisteremo su quelli che consideriamo i punti forti e anche i punti deboli del linguaggio.*

Per leggere questo libro è necessaria una buona conoscenza di uno o più linguaggi evoluti di programmazione. Si consiglia vivamente la conoscenza del Pascal e se conoscete un linguaggio "antico" come il Fortran o il Cobol vi consigliamo vivamente di cominciare leggendo un libro sul Pascal. Ad esempio "Programmare in Pascal" del Gruppo Editoriale Jackson.

In effetti, ADA è fortemente ispirato al Pascal: ha circa le stesse costruzioni di programmazione strutturata e, soprattutto, si pone nella stessa problematica per quanto concerne i tipi di dati. Introduce anche, naturalmente, nozioni completamente nuove (o prese da altri linguaggi).

Il primo capitolo dà una panoramica generale di ADA mostrando come i suoi concetti si agganciano a quelli del Pascal, li estendono, o li generalizzano o talvolta colmano certe lacune del Pascal. Questo capitolo fornisce infine l'elenco delle grandi novità di ADA rispetto al Pascal.

I capitoli seguenti riprendono una per una, approfondendole, le nozioni delineate precedentemente. Le grandi articolazioni sono;

— *le strutture* IF THEN ELSE , LOOP , CASE ...
— *i tipi*
— *i package*
— *il parallelismo e i task*
— *gli elementi generici*
— *i pragmi*

Gli input-output formano in ADA un package particolare descritto dopo l'introduzione dei package.

Infine, si conoscono già due forme di ADA (Ada preliminare e Ada rivisto). Noi descriviamo, naturalmente, la seconda versione, ma faremo qualche allusione ad Ada preliminare quando sarà necessario per chiarire un concetto.

Un certo numero di esempi di questo libro sono originali. Altri erano troppo classici o troppo dimostrativi per non essere stati già descritti.

Vogliamo ringraziare M. Jean ICHBIAH che ha cortesemente letto e commentato il manoscritto.

CAPITOLO 1

SGUARDO GENERALE SUL LINGUAGGIO: DAL PASCAL AD ADA

Il quaderno delle caratteristiche generali del Ministero della Difesa americano consigliava ai progettisti di ispirarsi al Pascal, Algol 68 o PL/1.

È significativo che i quattro progetti finalisti si siano tutti ispirati al **Pascal.**

Lo scopo di questo capitolo è di dare una visione generale di ADA sviluppando da un lato i concetti che sono il prolungamento di quelli del Pascal, di cui talvolta superano certe limitazioni, e dall'altro i concetti che appartengono ad ADA.

Le istruzioni di organizzazione dei programmi sono molto simili al Pascal, specialmente IF ... THEN ... ELSE e CASE. C'è anche ELSE IF (altrimenti..se), WHILE si scrive WHILE ... LOOP.

Tutte le strutture finiscono con un END corrispondente: END IF , END CASE, END LOOP ...

Ecco qualche esempio che ci darà nello stesso tempo un'idea dello "stile" di scrittura in ADA:

```
——calcolo di A come valore assoluto di B,
—— con un IF a un solo ramo:
        A:=B;
        IF B <0 THEN
                A:=—B;
        END IF;
```

—— con un IF simmetrico:

```
IF B<Ø THEN
          A:=—B;
        ELSE
          A:=B;
END IF;
```

Ritorneremo in seguito sulle regole di scrittura. Dagli esempi risulta che, come in Pascal, l'attribuzione si scrive :=. Questi esempi illustrano due cose importanti:

□ *la convenzione di scrittura* che noi impiegheremo nel testo:

1 — le **parole-chiave** del linguaggio saranno scritte in maiuscolo sottolineate (Come IF o THEN)
2 — le identificazioni scelte dal programmatore o predefinite nel linguaggio saranno maiuscole non sottolineate (Come A o B)
3 — le definizioni generali saranno in maiuscolo:

Esempio IF condizione THEN istruzione; END IF;
PROCEDURE nome IS testo della procedura;
END nome;

Nel primo esempio "condizione" deve essere sostituito da una scrittura di condizione corretta in ADA, "istruzione" da una istruzione ADA corretta:

```
IF A>B THEN A:=A+1; END IF;
```

Nel secondo, "nome" dovrà essere sostituito da un nome di procedura corretto scelto dal programmatore:

```
PROCEDURE TOTO IS ...
```

Naturalmente, nella battitura di un programma reale, le sottolineature non ci saranno.

□ *Il ruolo del punto e virgola è diverso* in ADA e nel Pascal: in ADA il punto e virgola è un semplice terminatore di istruzione. Vi sono quindi meno precauzioni per il suo uso: si mette alla fine di ogni istruzione, ed è tutto.

□ *Infine i commenti* sono introdotti in ADA da un doppio segno meno. Un commento termina la riga; può essere dietro una istruzione, ma non tra due istruzioni su una riga. Noi utilizzeremo in questo libro la stessa convenzione per commentare i nostri esempi:

```
A:=A+1;-- si aggiunge 1
```

Ritorniamo ad alcune particolarità delle istruzioni di strutturazione (lo studio completo sarà fatto nel prossimo capitolo):

L'istruzione LOOP permette di stabilire un loop indefinito:

```
LOOP
        istruzioni
END LOOP;
```

Si può uscirne perché esiste una istruzione EXIT sia diretta, sia sotto la forma EXIT WHEN condizione;

Questo permette di simulare il REPEAT UNTIL che non esiste in ADA:

```
LOOP
        Istruzioni;
EXIT WHEN condizione;
END LOOP;
```

A un livello più elevato, il programma può essere strutturato in blocchi che definiscono i loro propri dati,in procedure o funzioni. Questo aspetto del linguaggio verrà trattato più estesamente in seguito.

PRAGMI DICHIARAZIONI ISTRUZIONI ESEGUIBILI

È nota la classificazione che prevale nei linguaggi evoluti abituali tra istruzioni eseguibili e istruzioni non eseguibili o dichiarazioni. Questa classificazione esiste anche nei linguaggi assemblatori dove il ruolo delle istruzioni non eseguibili è dato dalle **direttive** che orientano il lavoro dell'assemblatore.

Nel Pascal le dichiarazioni hanno assunto un grande peso dovuto alla ricchezza dei tipi di dati trattati. Questa caratteristica è ancora più accentuata in ADA. In effetti le definizioni che si possono attribuire ai dati sono così sofisticate che incontrare una dichiarazione implica da parte del

compilatore un vero e proprio trattamento: si chiama **"l'elaborazione"** della dichiarazione. Inoltre l'elaborazione interviene in parte alla compilazione, in parte durante l'esecuzione.

È per questo che ci sono in ADA tre livelli: le *istruzioni eseguibili*, le *dichiarazioni*, e delle istruzioni che sono delle semplici direttive indirizzate al compilatore. Queste direttive si chiamano *pragmi*.

Esempio: PRAGMA OPTIMIZE(SPACE) domanda al compilatore di effettuare una ottimizzazione del codice generato per economizzare dello spazio in memoria.

GESTIONE DEI TIPI DI DATI

ADA è, ancora più del Pascal, un linguaggio fortemente tipizzato: ogni dato ha un tipo e non si può attribuire ad una variabile di un tipo un valore di un altro tipo.

Il Pascal è il primo linguaggio che, nel suo modo di definire i tipi di dati, ha fatto un passo verso il concetto di astrazione dei dati. Questo concetto permette al programmatore di definire i suoi dati in funzione delle proprietà voluta e non in funzione di questa o quella rappresentazione interna numerica che ne sarà fatta.

Per esempio, in Pascal, se volete trattare i giorni della settimana, definite il tipo GIORNO con:

```
TYPE GIORNO=(LUNEDÌ,MARTEDÌ,MERCOLEDÌ,GIOVEDÌ,
                  VENERDÌ, SABATO, DOMENICA);
```

In Basic, Fortran o PL/1 sareste obbligati a stabilire una corrispondenza (o codice) del tipo 1 ↔ lunedì,2 ↔ martedì...

Avendo definito il tipo di giorno e dichiarato:

VAR TODAY:GIORNO; siete al riparo da una assegnazione come TODAY:=GENNAIO; perché il compilatore ne verifica la conformità. Con un codice numerico potete sempre assegnare un valore; il compilatore non ha alcun mezzo di verificarne la validità.

Certamente con le dichiarazioni qui sopra potete scrivere:

```
IF TODAY=MARTEDÌ THEN...
```

o

```
FOR TODAY:=LUNEDÌ TO VENERDÌ DO ...
```

Ma il Pascal non va molto più lontano di questo, il che restringe terribilmente l'utilità di questa definizione dei tipi. Così le due dichiarazioni TYPE GIORNO= ... come sopra; WEEKEND =(SABATO, DOMENICA); sono incompatibili in quanto danno a SABATO e DOMENICA due tipi alla volta.

Inoltre i tipi intervallo di Pascal hanno certe limitazioni, come anche il trattamento dei numeri reali. ADA dà la soluzione a questi diversi problemi grazie alla nozione di tipi derivati, di sottotipi e di sovraccarico.

Tipi derivati:

TYPE T IS NEW S; definisce il tipo T a partire da S. T possiede tutte le literal di S e tutte le operazioni predefinite su S. È tuttavia distinto e, talvolta, certe conversioni sono necessarie. Notate che in ADA, le conversioni si fanno utilizzando il nome del tipo:

```
X:T; —— le dichiarazioni di variabili non
Y:S; —— necessitano la parola chiave VAR
X:=T(Y);
```

Esempio: supponete di aver definito un tipo colore:

```
TYPE COLORE IS (VERDE, ARANCIONE, ROSSO);
```

e voler particolarizzare un tipo colore di semaforo:

```
TYPE COLORE DI SEMAFORO IS NEW COLORE;
```

Un tipo derivato può essere unito a una condizione di intervallo:

```
TYPE INT IS NEW INTEGER RANGE —1000..+1000;
```

Si possono avere più tipi derivati con intervalli o valori che si ricoprono: è quello che si chiama il **sovraccarico** e si possono sempre eliminare delle ambiguità distinguendo (rispetto all'esempio qui sopra): COLORE (VERDE) da COLORE DI SEMAFORO (VERDE).

Sottotipi

Un sottotipo non fa che introdurre un vincolo — per esempio un vincolo di intervallo — su un tipo precedentemente definito. Ma gli elementi del

sottotipo conservano tutte le caratteristiche del tipo di partenza e non va prevista una conversione.

Esempio: SUBTYPE POSITIVO IS INTEGER RANGE
1... INTEGER'LAST;

Questo esempio ci fa scoprire gli **attributi**; ogni tipo T definisce il suo primo elemento chiamato T'FIRST, il suo ultimo chiamato T'LAST. L'elemento X del tipo T ha, nel tipo T, un successivo T'SUCC(X) e un precedente T'PREC(X).

TRATTAMENTO DEI REALI

I soli tipi numerici standard del Pascal sono INTEGER e REAL e noi abbiamo mostrato (D.J. DAVID e J.L. DESCHAMPS: Programmare in Pascal — Gruppo Editoriale Jackson) che questa impossibilità di modulare la precisione dei numeri era un fattore importante di **non-portabilità** del linguaggio.

Altri linguaggi offrono soluzioni; il FORTRAN permette di scegliere tra precisione semplice e doppia, il PL/1 permette di chiedere il numero di cifre significative volute.

ADA combina queste due soluzioni. Possiede i tipi standard INTEGER e FLOAT ma permette anche i tipi distinti SHORT INTEGER, LONG INTEGER, SHORT FLOAT e LONG FLOAT. Si raccomanda di non utilizzare direttamente uno di questi tipi standard ma di scrivere:

TYPE REAL IS NEW FLOAT;

e se per caso sulla macchina considerata il tipo FLOAT non vi dà una precisione sufficiente, non avete che da scrivere questa istruzione:

TYPE REAL IS NEW LONG FLOAT;

Ma la scelta può essere lasciata a discrezione del compilatore grazie a una soluzione di genere PL/1:

TYPE REAL IS DIGITS 10;

qui è il compilatore che sceglierà il tipo di base che assicurerà almeno dieci cifre di precisione.

Infine, si possono definire in ADA dei tipi intervallo anche sui reali (cosa impossibile nel Pascal) dando il passo di variazione desiderato:

```
TYPE NUMERO IS DELTA 0.01 RANGE 10.0 .. 100.0
```

definisce i numeri da 10 a 100 con passo 0.01 (tuttavia questo tipo non può servire da indice, né da variabile di loop).

TIPI COMPOSTI

Anche qui ADA colma una grave lacuna del Pascal: l'impossibilità di disporre di array di dimensioni variabili.

La dichiarazione di un array può essere **vincolata**:

```
TYPE VETTORE IS ARRAY (INTEGER RANGE 1..3) OF FLOAT;
```

o **non vincolata**:

```
TYPE MATRICE IS ARRAY (INTEGER RANGE < >, INTEGER
RANGE <>)) OF FLOAT;
```

Il tipo MATRICE può allora essere il tipo di un parametro formale di un sotto programma, nel qual caso le vere dimensioni saranno definite dall'argomento effettivo di chiamata. In alternativa, i limiti sono stabiliti quando si assegna il tipo MATRICE a un oggetto con un vincolo sull'indice o vengono dedotti dal valore inziale dell'oggetto:

```
M:MATRICE (1..2,1..3);
N:COSTANTE MATRICE:=((0.0,0.0),(0.0,0.0));
```

Si può anche introdurre un sottotipo:

```
SUBTYPE MATRICE 2 2 IS MATRICE (1..2,1..2);      poi
N:MATRICE 2 2;
```

La seconda soluzione del problema delle dimensioni variabili viene dall'alto grado di parametrizzazione del linguaggio che si ritroverà nei

RECORD parametrizzati. Anche in tipo vincolato si può scrivere:

```
TYPE TABELLA IS ARRAY (1..N) OF FLOAT;
```

con N variabile (che deve, naturalmente, avere un valore al momento dell'elaborazione della dichiarazione); si dice che si ha un array dinamico.

STRINGHE DI CARATTERI

ADA conosce il tipo predefinito CHARACTER formato da 128 caratteri ASCII:

```
TYPE CHARACTER IS (NUL, SOH,....,'A','B',.....,DEL)
```

I caratteri stampabili sono messi tra apostrofi, mentre i caratteri di controllo sono designati da un identificatore, per esempio, CR o LF.

Il tipo predefinito STRING è definito da:

```
TYPE STRING IS ARRAY(NATURAL RANGE <>)) OF CHARACTER;
```

dove NATURAL è il sottotipo predefinito:

```
SUBTYPE NATURAL IS INTEGER RANGE 1..INTEGER'LAST;
```

Si può allora definire una costante stringa con:

```
X:CONSTANT STRING:="BUONGIORNO"
```

dove

```
X:CONSTANT STRING:=('B','U','O','N','G','I','O','R','N','O');
```

e X ha ora per limiti 1..10. Ogni vettore ha una lunghezza reale che, per X, si scriverà X'LENGTH. ADA risolve i problemi di trattamento delle stringhe che creavano difficoltà in Pascal Zürich, tanto più che esiste un operatore di concatenazione (&) e che si può estrarre una parte di una stringa: X(2..4) è "UON" se X è la stringa sopra definita.

I RECORD

I RECORD in ADA sono più o meno come nel Pascal. Ci sono tuttavia due miglioramenti:

Possibilità di definire dei valori iniziali per default

```
TYPE COMPLEX IS RECORD
        RE,IM:FLOAT:=0.0;
        END RECORD;
```

assicura che ogni numero complesso che si definisce è inizializzato a zero salvo che si specifichi un altro valore.

Tipi parametrizzati

C'è un miglioramento dei RECORD con varianti del Pascal.

```
TYPE GENERE IS (M,F);
TYPE PERSONA (SESSO:GENERE) IS
        RECORD
           DATANASC:DATA;—— il tipo DATA è supposto definito prima
              CASE SEXE IS —— vedere avanti la sintassi di CASE
                 WHEN M => BARBU:BOOLEAN;
                 WHEN F => NUMERO-BAMBINI:INTERO;
              END CASE;
        END RECORD;
```

Dichiarazioni possibili a partire di qui:

```
SUBTYPE DONNA IS  PERSONA (F);
GIULIA:DONNA:=(F,(10,DIC,1980),0);
GIANNI:PERSONA(M):=(M,(15,MAG,1950),FALSE);
```

Si può dare un valore iniziale di default al discriminante:

```
TYPE PERSONA (SESSO:GENERE:=F) IS ...
```

Questa parametrizzazione offre un'altra soluzione al problema delle

stringhe di lunghezza variabile:

```
TYPE CHAINE (LMAX:INTEGER RANGE 1..256) IS
   RECORD
      LUNGHEZZA:INTEGER RANGE 0..256:=0;
      CONTENUTO :ARRAY (1..LMAX) OF CHARACTER;
   END RECORD;
```

I SOTTOPROGRAMMI

Arriviamo ora all'introduzione di nuovi concetti. ADA, come il Pascal, permette le funzioni e le procedure. I parametri (argomenti) inviati e recuperati sono ancor meglio distinti: ci son tre modi IN, OUT e IN OUT:

```
PROCEDURE INSERZIONE (V:IN INTEGER; INTO: IN OUT PTR) IS
      T:PTR;
   BEGIN
      T:=NEW OGGETTO (V,INTO);
      INTO:=T;
   END INSERZIONE;
```

è una procedura di inserzione dell'informazione V nell'oggetto di tipo OGGETTO puntato da INTO (per formare una lista).

Questo mostra che i tipi puntatore sono trattati in modo analogo al Pascal. Si avrebbero, per l'esempio sopra, le dichiarazioni:

```
TYPE OGGETTO; —— dichiarazione incompleta
TYPE PTR IS ACCESS OGGETTO;
TYPE OGGETTO IS
   RECORD
      VALORE:INTEGER;
      SEGUENTE:PTR;
   END RECORD;
```

Un oggetto sarà creato per esempio da T:=NEW OGGETTO (0, NULL) che crea un oggetto di valore zero, senza successore e puntato da T che è proprio del tipo PTR.

Le particolarità di procedura di ADA sono:

1 — la possibilità di valore di default dati ai parametri IN. Il parametro può allora essere omesso alla chiamata; il valore preso sarà quello di default.

2 — ci si può ricordare il nome di un parametro formale durante la chiamata:

```
INSERZIONE(5,INTO=>LISTA);
```

3 — **il sovraccarico:** si è visto che parecchi tipi enumerativi possono contenere la stessa costante (designata dal suo nome). Analogamente, parecchi sottoprogrammi con lo stesso nome possono coesistere a uno stesso livello di chiamata: questo indipendentemente dalle diverse versioni di un identificatore definito in un blocco, purchè gli argomenti siano diversi, almeno per il tipo. Questo ci porta a sviluppi interessanti. Innanzitutto si possono **sovraccaricare gli operatori aritmetici classici**. Questo colma una lacuna molto importante del Pascal. Nel Pascal si può perfettamente definire il tipo complesso:

Pascal

```
TYPE COMPLEX=
 RECORD
          RE,IM:REAL
 END;
```

ADA

```
TYPE COMPLEX IS
    RECORD
         RE,IM:FLOAT;
      END RECORD;
```

Si vede la similitudine di scrittura. Ma se in Pascal abbiamo
VAR ZI,Z2,Z3:COMPLEX
non si può scrivere Z3:=Z1+Z2; Pascal si ferma a metà del cammino; infatti permette di definire il tipo COMPLEX ma non di dire cosa è l'addizione di due complessi!

In Pascal bisognerebbe definire una funzione:

```
FUNCTION SOMMEC (A,B:COMPLEX):COMPLEX;
   BEGIN
      SOMMEC.RE:=A.RE+B.RE
      SOMMEC.IM:=A.IM+B.IM
   END
```

e scrivere:
Z3:=SOMMEC(Z1,Z2);

In ADA, noi sovraccarichiamo l'operatore +:

```
FUNCTION "+" (X,Y:COMPLEX) RETURN COMPLEX IS
  BEGIN
    RETURN (X.RE+Y.RE,X.IM+Y.IM);
  END "+";
```

e scriviamo Z3=Z1+Z2; ADA "saprà" che qui c'è da applicare l'addizione complessa. Ritorneremo su questo esempio a proposito dei package.

Una possibilità molto interessante di sovraccarico sarà, per esempio, definire:

```
FUNCTION "*" (X,Y:MATRICE) RETURN MATRICE IS
```

per programmare ad esempio il prodotto di matrici.

Si potrà allora, altrettanto semplicemente che in APL, scrivere un prodotto di matrici sotto la forma:

```
C:=A*B;
```

TIPI GENERICI

Le operazioni sono, lo si è visto, definite su operatori di tipo determinato. Un errore viene segnalato (si dice in ADA che si tratta di una EXCEPTION) quando si cerca di applicarli ad operatori di altro tipo. Si può rendere un operatore applicabile a tutti i tipi definendo dei sovraccarichi per ogni combinazione di tipi suscettibili di presentarsi, ma si tratta di un procedimento tedioso.

Si può scrivere, ad esempio, per una procedura di scambio di due oggetti:

```
GENERIC TYPE T IS PRIVATE
 PROCEDURE EXCHANGE (X,Y): IN OUT T) IS
    Z:T;
    BEGIN
     Z:=X;X:=Y;Y:=Z;
    END SCAMBIO;
```

Per utilizzarla per scambiare due matrici bisognerà, come si dice, circostanziarla con:

```
PROCEDURE SCAMBMAT IS NEW SCAMBIO (T=>) MATRICE):
```

poi si potrà scrivere:

```
SCAMBMAT (A,B);
```

I PACKAGE

Ecco infine uno degli elementi più potenti introdotti da ADA. Generalizzando quello che abbiamo visto sopra, noi potremmo fare tutto un sistema di trattamento dei numeri complessi. Questo in ADA si chiama un **package**. Il package è composto di due parti: una **definizione** dove si enumera ciò che c'è nel package e se ne descrivono le proprietà, e un **corpo** del package, dove le funzioni sono realizzate. Per esempio noi potremmo avere:

```
PACKAGE NUMERI COMPLESSI IS
    TYPE COMPLEX IS PRIVATE;
    I: CONSTANT COMPLEX;
    FUNCTION "+" (X,Y:COMPLEX) RETURN COMPLEX;
    FUNCTION "—" (X,Y:COMPLEX) RETURN COMPLEX;
    FUNCTION "*" (X,Y:COMPLEX) RETURN COMPLEX;
    FUNCTION "/" (X,Y:COMPLEX) RETURN COMPLEX;
    FUNCTION CMPLX(RP,IP:FLOAT) RETURN COMPLEX;
    FUNCTION REAL (X:COMPLEX)RETURN FLOAT;
    FUNCTION IMAG(X:COMPLEX) RETURN FLOAT;
PRIVATE
    TYPE COMPLEX IS
     RECORD
      RE:FLOAT;
      IM:FLOAT;
    END RECORD
  I: CONSTANT COMPLEX:=(0.0,1.0);
END NUMERI COMPLESSI;
```

Il corpo del package sarà della forma:

```
PACKAGE BODY NUMERI COMPLESSI IS
  FUNCTION "+"(X,Y:COMPLEX) RETURN COMPLEX IS
    BEGIN
     RETURN  (X.RE+Y.RE,X.IM+Y.IM);
    END "+";

   .
  FUNCTION "*" (X,Y:COMPLEX) RETURN COMPLEX IS
    BEGIN
     RETURN (X.RE*Y.RE—X.IM*Y.IM,
     X.RE*Y.IM+X.IM*Y.RE);
    END "*";

   .
  FUNCTION CMPLX(RP,IP:FLOAT') RETURN COMPLEX IS
    BEGIN
     RETURN (RP,IP):
    END CMPLX;

   .
  FUNCTION REAL(X:COMPLEX) RETURN FLOAT IS
    BEGIN
     RETURN X.RE;
    END REAL;
   .

   .
  END NUMERI COMPLESSI;
```

Un modo di utilizzarli in un blocco sarebbe:

```
DECLARE —— introduce la parte dichiarativa del blocco
  USE NUMERI COMPLESSI;
  A,B:COMPLEX;
  C,D:FLOAT;
```

```
BEGIN —— introduce la parte esecutiva del blocco
   A:=CMPLX(1.0,—5.0);—— numero 1-5i
   B:A+I;                 —— addizione complessa
   D:=1.0;
   C:=REAL (B)+D;         —— addizione reale
END;
```

I DATI PRIVATE

Ci resta da spiegare la denominazione PRIVATE dell'esempio sopra. In breve è il modo in cui ADA ottiene che ciascuno si occupi di quello che lo interessa e non del resto. Quello che interessa l'utilizzatore, è di sapere che c'è un tipo COMPLEX, che i complessi possono sommarsi, moltiplicarsi, ecc... Gli è indifferente conoscere il modo in cui viene ottenuto il risultato. È per questo che la definizione esatta del tipo complesso è PRIVATE. E se il realizzatore della biblioteca matematica avesse voluto gestire numeri complessi in coordinate polari, avrebbe scritto:

```
TYPE COMPLEX IS
   RECORD
   R:FLOAT;
   THETA:FLOAT;
  END RECORD
I: CONSTANT COMPLEX:=(1.0,PI/2.0);—— si suppone che PI sia
definito.
```

e avrebbe riscritto le funzioni, per esempio:

```
FUNCTION "*" (X,Y:COMPLEX) RETURN COMPLEX IS
  BEGIN
   RETURN (X.R*Y.R,X.THETA+Y.THETA);
  END;
```

ma questo non avrebbe cambiato **nulla** per il programma utente.

Questa caratteristica di ADA è molto importante: essa è un passo di più verso l'astrazione dei dati che permette al programmatore di definire i suoi dati in base alle proprietà che essi devono avere, tenuto conto delle applicazioni da trattare, e non in base alla loro rappresentazione interna.

I package sono del resto uno degli elementi più utili in ADA; quando si vuole mettere a disposizione un insieme di risorse (tipi di dati, sottoprogrammi....) legati logicamente, si costituisce un package. Per esempio i procedimenti di input-output standard formano un package chiamato INPUT-OUTPUT. Per il momento noi diremo che queste due procedure essenzialmente sono:

GET (variabile) che legge la variabile dalla tastiera e
PUT (espressione) che scrive il valore dell'espressione sullo schermo.

Con sovraccarichi opportuni, GET è capace di leggere dalla tastiera un valore appartenente a un tipo enumerativo definito dall'utilizzatore. Se si ha:

```
TYPE GIORNO IS (LUNEDÌ, MARTEDÌ...);
TODAY:GIORNO;
```

e se, in risposta a GET(TODAY); si batte la stringa di caratteri MARTEDÌ, il sistema attribuirà a TODAY il valore MARTEDÌ.

Si sa che il Pascal è **totalmente incapace** di questo (le sue procedure di input-output non accettano altro che tipi standard), il che limita molto l'interesse del tipo utente nel Pascal, mentre ADA esplora tutte le possibilità desiderabili.

La sola apparente carenza in ADA rispetto al Pascal, nel dominio dei tipi, è l'assenza di SET. Bisogna procedere per tabelle Booleane.

ECCEZIONI, TASK, PARALLELISMO

Abbiamo visto ora gli aspetti di ADA che derivano dalla problematica di gestione dei dati introdotta dal Pascal. ADA presenta altri aspetti che saranno studiati nel seguito del libro. Tuttavia, menzioneremo le eccezioni (trattamento di errori) e i task (trattamento in parallelo) affinché la nostra rassegna sia quasi completa.

Eccezioni

Un certo numero di errori standard fanno scattare una eccezione predefinita come NUMERIC-ERROR. Altre eccezioni possono essere definite

dall'utente:

```
MATRICE SINGOLARE: EXCEPTION;
```

Una eccezione può essere generata di forza da RAISE:

```
IF D=0.0 THEN RAISE MATRICE SINGOLARE;
```

Il trattamento delle eccezioni è dato sotto la forma:

```
EXCEPTION
    WHEN MATRICE_SINGOLARE=> istruzioni opportune.
```

I task

I task definiscono dei trattamenti che possono essere eseguiti in parallelo; ADA non impone la simultaneità, sono le risorse della macchina che eventualmente lo permettono.

Per questo ADA permette di definire dei task che devono essere tutti completati per permettere la fine del programma chiamato. Supponiamo, per esempio, che una famiglia sbarchi ad un aeroporto. Bisogna recuperare il bagaglio, affittare una automobile e fissare un albergo. Le tre cose possono essere fatte simultaneamente se il padre si occupa dell'hotel, la madre affitta l'auto e i bambini si occupano dei bagagli. Non si lascerà l'aeroporto che quando tutto sarà fatto. Si può scrivere:

```
PROCEDURE ARRIVO_AEROPORTO IS
  TASK RECUP_BAGAGLI IS
  .
  .
  END ;
  TASK BODY RECUP_BAGAGLI IS
  .
  .
  END RECUP_BAGAGLI;
TASK AFFITTO_VETTURA;...; END;
  TASK BODY AFFITTO_VETTURA IS;
  .
```

```
 END AFFITTO_VETTURA;
BEGIN
 FISSARE_HOTEL;
END ARRIVO_AEROPORTO;
```

La sincronizzazione può essere affinata con le istruzioni di appuntamento ACCEPT e ENTRY che vedremo nel capitolo dedicato ai task.

Ecco terminato questo colpo d'occhio sul linguaggio ADA. È stato rapido su certi aspetti su cui si entrerà ora in dettaglio ma speriamo abbia dimostrato:

— la grande potenza e il carattere completo di ADA,
— come ADA si colloca nel prolungamento del Pascal e come ne colma in modo soddisfacente le lacune più gravi.

CAPITOLO 2

ISTRUZIONI ESEGUIBILI E STRUTTURE

REGOLE GENERALI DI SCRITTURA — IL SET DI CARATTERI ADA

Ogni programma ADA può esprimersi grazie all'aiuto del **set di caratteri di base** formato.

— dalle lettere maiuscole A...Z
— dalle cifre 0...9
— dai caratteri speciali "#%&'()*+,—./:;<=>_ | più lo spazio

È bene distinguere — (segno meno o trattino) dal _ (sottolineato), lo 0 (zero) da O.

Non useremo in questo libro alcuna convenzione consistente nello sbarrare l'uno o l'altro: la distinzione avverrà sulla base del contesto.

Si considera anche un set di caratteri esteso che comprende oltre ai caratteri precedenti:

— le lettere minuscole a...z
— altri caratteri speciali ! $? [\] ^ '{} ~

Ogni programma può essere convertito nel set ridotto; una lettera minuscola è l'equivalente della maiuscola corrispondente salvo nella stringa di caratteri. Essa può allora essere espressa sotto la forma ASCII. LC_X (per la x). Anche $ può essere rimpiazzato da ASCII.DOLLAR, ecc....
Non approfondiremo ulteriormente l'argomento, limitandoci al set di base.

Con questi caratteri, si formano delle parole (o unità lessicali).

Le parole sono:

— le parole-chiave riservate al linguaggio (esempio IF PROCEDURE)
— gli identificatori predefiniti o designati dal programmatore (esempio) PUT, A1234)

— le literal numeriche o alfanumeriche (esempio 0.456, 'A', "ZOZO")
— i commenti (esempio: —— QUESTO È UN COMMENTO
— i delimitatori: &'()*+,-./:;<=>1
o le combinazioni => .. ** := /= >= <= << >> <>

Due parole possono (e più spesso devono) essere separate da almeno uno spazio bianco o un ritorno a capo.

Non deve al contrario esservi dello spazio né ritorno a capo nel mezzo di una parola (tranne se si tratta di una literal alfanumerica).

Diversamente, la suddivisione della scrittura sulla linea è libera; le differenti istruzioni sono terminate da punti e virgola. Si raccomanda pertanto che ciascuna istruzione corrisponda ad una linea e la cosa migliore è curare l'impaginazione in modo che faccia risaltare la struttura del programma.

In PASCAL o in PL/1, è possibile inserire un commento ovunque si possa mettere uno spazio.

ADA è più restrittivo. Si può iniziare un commento alla fine di qualsiasi parola, ma questo deve terminare la linea; la struttura:

inizio d'istruzione —— commento seguito dell'istruzione

è impossibile perché se esiste un segnale di inizio del commento (il doppio trattino), non esiste un segnale di fine.

La forma più normale sarà:

istruzione; —— commento

Un commento può, sicuramente, essere da solo sulla linea:

—— INIZIO DELLE ITERAZIONI

Una riga di trattini rappresenta un caso particolare (a causa dei due primi trattini) e può portare ad una buona impaginazione, separando bene i moduli:

```
——
——
——
——————————————————————————————
——
——   SOTTO-PROGRAMMA DI...
——
——
```

STRUTTURA GENERALE DI UN PROGRAMMA

Tralasciamo qui i problemi di compilazione separata che saranno trattati in un capitolo a parte.

Ogni programma si presenta come un insieme di unità di compilazione (procedure, package o compiti (task)) o di blocchi.

Queste unità sono annidabili. Una procedura può contenerne altre, un compito può inglobare delle procedure e così via.....

Ciascuno di questi moduli consta di due parti ben distinte:

— **la parte dichiarativa** introdotta dall'intestazione del modulo (PROCEDURE .... PACKAGE ...... o TASK .....) o da DECLARE se è un semplice blocco.
— **la parte esecutiva** che inizia con BEGIN e termina con END.

La parte dichiarativa è formata dalle dichiarazioni che definiscono i dati che saranno manipolati nel modulo.

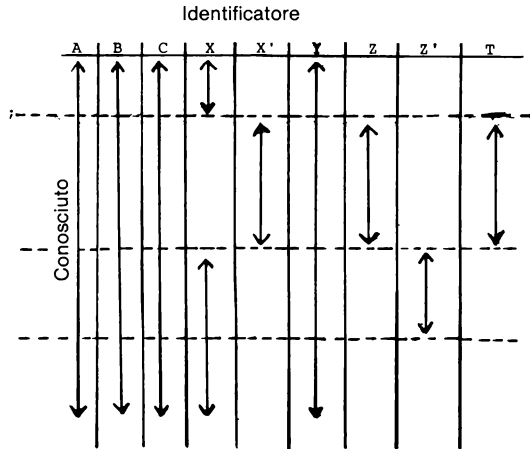
La parte esecutiva contiene le istruzioni eseguibili che rappresentano il trattamento propriamente detto.

PORTATA DEGLI IDENTIFICATORI

Le regole di portata degli identificatori (si chiama portata di un identificatore la zona di programma in cui è conosciuto) sono ispirate a quelle del Pascal. Le citiamo qui sotto in prima approssimazione e saranno riprese completamente nel capitolo IV.

La regola generale è che un oggetto è conosciuto nel modulo in cui è definito e nei moduli che vi sono contenuti (appare dunque come globale). Però non è conosciuto nei moduli esterni al modulo di definizione o nel modulo che ingloba quello di definizione (appare dunque come locale).

Infine, se un modulo A dichiara una variabile X e contiene un modulo B, anche B può dichiarare X, e a questo punto, questa X — che chiameremo X' — è distinta dalla prima X. La prima X non è accessibile in B (e i moduli contenuti). Si dice che è "nascosta".

Esempio: le dichiarazioni e gli annidamenti della pagina successiva danno luogo allo schema:

ESERCIZIO 2.1: Fare lo stesso schema per l'annidamento:

```
PROCEDURE  A;
     X,Y,Z:...;
   PROCEDURE  B;
       X:...;
        PROCEDURE  C;
          Y:...;
           PROCEDURE  D;
             X:...;
              BEGIN
               .
               .
               .
              END  D;
           BEGIN
             .
             .
             .
           END  C;
        BEGIN
          .
          .
          .
        END  B;
 BEGIN
   .
   .
   .
 END  A
```

GLI IDENTIFICATORI

Gli identificatori ADA comprendono le parole-chiave del linguaggio (Appendice 1), gli identificatori predefiniti, ma che il programmatore può ridefinire, e gli identificatori completamente definiti dal programmatore.

Gli identificatori possono designare principalmente delle costanti, delle variabili, dei tipi, delle procedure, dei package, delle eccezioni e dei task.

Tutti gli identificatori seguono le stesse regole. Il primo carattere è una lettera, i seguenti sono lettere o cifre o sottolineature (non devono esservi più sottolineature consecutive).

```
Esempio: ALFREDO
         NUMERO-DI-VIAGGIATORI
         H2SO4
         ENRICO-8
```

Il linguaggio non limita il numero dei caratteri e questi sono tutti significativi. Come di consueto, si consiglia di restare entro i limiti del ragionevole.

Oltre agli identificatori così definiti, le espressioni possono fare intervenire degli identificatori indicizzati (Esempio: M(5) è la quinta componente del vettore M), dei segmenti (Esempio: M(3...6) designa le componenti da 3 a 6 del vettore M), degli identificatori qualificati (Esempio: SYSTEM.MAX_INT designa la costante che rappresenta il massimo intero definito nel package SYSTEM), degli attributi (Esempio: ZOZO'SIZE designa il numero di byte occupati dall'oggetto ZOZO). Questi elementi saranno definiti più precisamente qui sotto.

LE LITERAL

Una costante può essere rappresentata sia da un nome simbolico (che è un identificatore), sia da una literal che abitualmente rappresenta la costante.

Le literal numeriche sono sia sotto forma decimale, che in forma base. La forma decimale è quella degli altri linguaggi:

```
254          -1609       (interi)
12.34        -0.459492   (virgola fissa)
0.1234E-8    -52.3E15    (virgola mobile)
```

La sola differenza è che ADA consente d'inserire delle sottolineature per migliorare la leggibilità:

```
1-619-754-321
3.141-592-65
```

La forma base permette ad ADA di esprimere molto comodamente i numeri in qualsiasi base; si scrive:

base#numero#
oppure
base #numero#E esponente

2#0111 —1111#O 16#7F# rappresenta 127
16#E#E1 rappresenta $14 \times 16^1 = 224$
16#FF.F#E1 rappresenta FFF16 = 4095
3#12#rappresenta 5

Le basi possono andare da 2 a 16. Quando la base è superiore a 10, si utilizzano le lettere ABCDEF come cifre di valore 10,11,12,13,14, e 15.

ESERCIZIO 2.2: Rappresentare il numero 154 in binario. Quanto vale la costante 12 #1b #?

Le literal alfanumeriche sono sia dei caratteri isolati chiusi dentro apostrofi (Esempio: 'A', '*') che stringhe di caratteri chiusi fra virgolette (Esempio: "BUONGIORNO"). Come di regola, se si vuole includere una virgoletta nella stringa, la si raddoppia: "OGGI" "DI". " " è la stringa vuota.

La stringa deve occupare soltanto una riga altrimenti bisogna procedere per concatenazione (operatore &). Bisogna utilizzare la concatenazione e i nomi simbolici di caratteri se si vogliono incorporare dei caratteri di controllo nella stringa:

Esempio: "BUONGIORNO"&ASCII.CR termina con un "ritorno-a-capo".

ADA distingúe 'A' che è uno scalare da "A" che è una stringa di un solo carattere.

ISTRUZIONI SEQUENZIALI

Le istruzioni sequenziali effettuano le tre operazioni fondamentali di ogni trattamento informatico:

- □ l'acquisizione di dati
- □ i calcoli effettuati su questi dati
- □ l'uscita dei risultati

L'acquisizione e l'uscita dei dati si ottiene chiamando delle procedure che fanno parte di un package standard. Fino al capitolo 5 diremo provvisoriamente che si legge un dato mediante:

GET (nome di variabile);

e che si ottiene un risultato con PUT (espressione aritmetica);

Ci resta dunque da vedere l'istruzione fondamentale di trattamento che è quella di attribuzione della forma:

variabile := espressione aritmetica

Il segno di attribuzione è asimmetrico come in ogni linguaggio "pensato". Si valuta l'espressione aritmetica (che può ridursi ad una semplice variabile) ed il risultato è compreso in un range della variabile che compare a sinistra del segno :=.

Operandi:

Gli operandi delle espressioni aritmetiche possono essere delle literal, delle costanti (designate da un nome simbolico), delle variabili, delle chiamate di funzione.

Le variabili possono presentarsi sotto diverse forme:

Variabile semplice: M NUMERO_DI_PUNTI...

Variabile indicizzata: V(3) terza componente del vettore V
V(3*I+1)
MAT(5) (4) quarto elemento della quinta riga della matrice MAT
ECHIQUIER(12,12) altro modo di indicare un elemento di una matrice

Segmento: V(5...12) designa il vettore formato degli elementi da 5 a 12 del vettore V

V(5...12) (9) è identica a V (9)

Variabile qualificata: NASCITA. GIORNO elemento GIORNO della registrazione NASCITA

PTR.ALL oggetto puntato dal puntatore PTR (si scriverebbe PTR^ in Pascal)

SUB.X (variabile X dichiarata nella procedura SUB (supposta visibile, dunque inglobante il segmento in cui si trova; X da sola designa la X dichiarata più recentemente)

Attributi: Ogni tipo trasmette alle sue variabili certi attributi predefiniti. Eccone qualche esempio:

VAR'ADDRESS:	indirizzo del primo byte in cui è compreso l'intervallo della variabile VAR
T'FIRST:	valore minimo di un TYPE T: se si ha un TYPE COLORE IS (ROSSO, GIALLO, BLU), il COLORE'FIRST è ROSSO.
X'LENGHT:	numero di componenti del vettore X.

Chiamata di una funzione

ABS(X)

AGGREGATI

Il vincolo fondamentale dell'operazione di assegnazione è che i due termini devono essere rigorosamente dello stesso tipo.

Non si può collocare in una variabile un risultato che non sia dello stesso tipo.

Non esiste il concetto di tipi compatibili o di conversione implicita. Ogni

conversione deve essere esplicitamente indicata, ed è sufficiente fare intervenire il nome del tipo di arrivo.

Se si ha:

```
I:INTEGER;
R:FLOAT;
```

si può scrivere:

```
I:=INTEGER(R);
```

Ci sono delle condizioni di compatibilità perché la conversione sia possibile; in generale è sufficiente il buon senso.

Detto questo, l'assegnazione può concernere degli oggetti di un tipo strutturato, matrice o record. Se A e B sono dello stesso tipo, matrici delle stesse dimensioni, o record della stessa struttura, A:=B; è possibile. Ogni elemento di A è reso uguale all'elemento corrispondente di B.

In quest'ultimo caso, se il valore che si vuole variabile è una costante, bisognerà che sia una costante multipla, con un buon numero di elementi. Questo si chiama un aggregato.

Supponiamo di avere:

```
TYPE VECT5 IS ARRAY (1...5) OF INTEGER;
V:VECT5;
```

si potrebbe scrivere:

```
V:=(1,0,1,0,0); —— aggregato posizionale
```

si potrebbe ancora scrivere:

```
V:=(1!3=> 1,2=> 0,4...5 => 0);
```

o ancora:

```
V:=(1!3=> 1, OTHERS => 0);
```

(aggregato definito dai nomi delle componenti)

Altrettanto per un RECORD,

se si ha:

```
TYPE DATE IS
    RECORD
          GIORNO:INTEGER RANGE 1..31;
          MESE: (GEN,FEB,MAR,APR,MAG,GIU,LUG,AGO,SET,OTT,
          NOV,DIC);
          ANNO:INTEGER RANGE 1800...2000;
    END RECORD;
D:DATA;
```

si potrà scrivere:

```
D:=(15,AGO,1981);
```

oppure:

```
D:=(MESE=> AGO,GIORNO=>15,ANNO=>1961);
```

e nell'aggregato nominato, l'ordine degli elementi non conta poiché l'elemento è definito col suo nome.

ESERCIZIO 2.3: Assegnare la stessa data a D in un altro modo (che non sia un semplice cambiamento d'ordine).

OPERATORI

L'espressione di base in ADA è l'espressione aritmetica che indica un calcolo aritmetico da fare. Gli operatori corrispondenti sono:

```
**     (elevazione a potenza)
*      (moltiplicazione)
/      (divisione)
MOD    (modulo)      REM (resto (remainder))
+      (addizione)
—      (sottrazione o opposto)
```

L'ordine nel quale una espressione è valutata è determinato dall'ordine di priorità degli operatori, la regola da sinistra a destra e l'uso di parentesi.

L'ordine di priorità degli operatori è lo stesso che in Fortran o PL/1. In ordine di priorità decrescente, si ha **, poi ex equo *,/, MOD e REM (operatori moltiplicativi), poi gli operatori monadici + e —, poi infine + e — nel senso diadico.

Ricordiamo che un operatore monadico ha un solo operando posto alla sua destra. —X (prendere l'opposto di X) è monadico; in A—B, al contrario, — è diadico.

Quando si è in presenza di due operatori della stessa priorità, li si esegue iniziando da quello più a sinistra.

Tuttavia, A**B**C è vietato. Bisogna inserire delle parentesi: A**(B**C).

Infine, si può sempre imporre che una sottoespressione sia valutata per prima, con l'uso di parentesi.

Queste regole si riassumono definendo:

primario:	:= literal | variabile | chiamata di funzione | conversione di tipo | espressione qualificata | (espressione) (: :=si legge "consiste in"; | si legge "o")
fattore:	:=primario | primario ** primario
termine:	:=fattore | fattore termine operatore di moltiplicazione
espressione semplice:	:= termine | termine operatore di addizione espressione semplice
espressione aritmetica:	:=espressione semplice | operatore monadico espressione semplice

ESERCIZIO 2.4: Verificare che le definizioni qui sopra esprimano correttamente l'ordine di valutazione delle espressioni che sono state indicate.

Nota: il linguaggio non specifica l'ordine nel quale sono valutati i due operandi di un'operazione. Di conseguenza, bisogna che il significato dell'espressione non dipenda dal linguaggio, altrimenti sarebbe possibile commettere errori. I due operandi di un'operazione devono essere dello stesso tipo (e di un tipo per il quale l'operazione sia definita) oppure bisogna che l'operatore sia stato sovraccaricato per i tipi coinvolti.

LA CONCATENAZIONE

L'operatore & effettua la concatenazione (cioè l'unione di due stringhe di caratteri: "BUON" & "GIORNO"="BUONGIORNO"). Infatti, si applica ad ogni coppia di matrici unidimensionali di base dello stesso tipo. Dal punto di vista priorità, & viene ad affiancarsi agli operatori di addizione diadici + e —.

ESPRESSIONI LOGICHE

Si definiscono dapprima gli operatori di relazione che legano due espressioni aritmetiche. Il risultato è booleano, TRUE se la relazione è vera, FALSE se la relazione è falsa; per esempio: 3>5 è FALSE. Gli operatori sono: = (uguaglianza), /= (diverso: non < > come in Basic), <(minore), <= (minore o uguale), > (maggiore), >= (maggiore o uguale).
Tutti gli operatori di relazione hanno la stessa priorità e questa priorità è inferiore alla più debole priorità degli operatori aritmetici.

Di conseguenza, nella:

espr. aritm. 1 operatore di relazione espr. aritm. 2
si valutano dapprima le due espressioni aritmetiche poi si valuta se la relazione indicata è soddisfatta.

Se si vogliono combinare più relazioni, è obbligatorio fare uso delle parentesi.

ESERCIZIO 2.5: Valutare (1<5) <(2>3). Quale osservazione fate su: 1<(5<2)>3

RELAZIONE D'APPARTENENZA

Gli operatori IN e NOT IN hanno la stessa priorità degli operatori di relazione e forniscono anche loro un risultato booleano. NOT IN rappresenta il contrario di IN. IN è della forma:

elemento IN intervallo, esempio: GIORNO IN LUNEDÌ.. VENERDÌ

oppure

elemento IN sottotipo, esempio: K IN NATURAL

Il risultato è TRUE se il valore di K risponde ai requisiti che definiscono il sotto-tipo.

OPERATORI LOGICI

Le relazioni possono essere combinate grazie a degli operatori logici per formare delle espressioni logiche (di risultato booleano).

Gli operatori logici diadici hanno tutti la stessa priorità e quest'ultima è inferiore a quelle degli operatori di relazione.

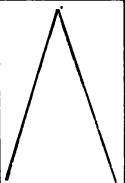
Dunque in:

relazione 1 operatore logico relazione 2

si valutano dapprima le due relazioni poi le si combina.
Ricordiamo che in ciascuna delle relazioni, si valutano dapprima le espressioni aritmetiche che intervengono.

Gli operatori sono: AND (e), OR (o), XOR (o esclusivo) e le forme veloci AND THEN (dunque) e OR ELSE (altrimenti).

Esse corrispondono alle tabelle di verità:

A	B	A AND B	A OR B	A XOR B
FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	FALSE

Le forme veloci permettono di evitare di calcolare la relazione 2 se, dopo l'analisi della relazione 1 si può subito dedurre quale sarà il risultato dell'operazione.

Per A AND THEN B, se A è FALSE, non si calcola B e si sa subito che il risultato è FALSE. Se A è TRUE, bisogna calcolare B. Se B è TRUE, il risultato è TRUE, se B è FALSE, il risultato è FALSE.

Per A OR ELSE B, se A è TRUE, non si calcola B e si sa subito che il risultato è TRUE. Se A è FALSE, bisogna calcolare B e il risultato è identico a B.

Esiste un operatore monadico NOT (che restituisce il contrario del suo operando) che ha la stessa priorità di + e di — monadici.

Da ultimo, si può costruire la tabella completa degli operatori che segue:

		priorità
logici	AND OR XOR	
relazioni	= /= < <= > >= IN NOT IN	
addizione	+ — &	
monadici	+ — NOT	
moltiplicazione	* / MOD REM	
esponenziazione	**	

Nota: Sebbene si possa applicare la regola del tutto a sinistra, ADA vieta che gli operatori logici siano mescolati senza parentesi.

ESERCIZIO 2.6: Non c'è l'operatore XOR. Come fare?

ESERCIZIO 2.7: Esprimere diversamente: (X>=3) AND (X<=10).

Abbiamo ora definito le operazioni aritmetiche utilizzabili. Saranno discusse nel capitolo seguente le particolarità legate a certe operazioni in funzione dei tipi di dati ai quali si applicano.

Il linguaggio in realtà non possiede che una funzione predefinita nel package STANDARD ed è ABS (valore assoluto), ma nulla impedisce che un programma o un utilizzatore ne definiscano delle altre. È evidente che per effettuare dei calcoli matematici, bisognerebbe definire delle funzioni come il seno, l'esponenziale, eccetera....

Dopo le istruzioni sequenziali (non abbiamo visto per il momento che la principale: assegnazione), vediamo le istruzioni di salto o di strutturazione dei programmi.

ISTRUZIONE GOTO

Sebbene strutturato, ADA possiede l'istruzione di salto incondizionato GOTO. Si cercherà di usarla il meno possibile.

Si esprime nella forma: GOTO etichetta;
GOTO LÀ;

nell'istruzione referenziata dall'etichetta, quest'ultima si presenta circondata da << >>

<<LÀ>> I:=I+1;

GOTO può essere utile per uscire da un ciclo in casi speciali (a meno che EXIT risolva meglio il problema).

GOTO non può, venendo dall'esterno, entrare in un'istruzione composta (blocco IF, CASE o ciclo) o in una PROCEDURE.

Non può "uscire" dal corpo di un sotto-programma, da un package, o da un task.

ISTRUZIONE IF

```
IF    condizione 1    THEN
    seguito delle istruzioni 1;
 ELSIF    condizione 2    THEN
    seguito delle istruzioni 2;
 ELSE
    seguito delle istruzioni 3;
 END   IF;
    seguito delle 4;
```

Le condizioni rappresentano delle espressioni booleane generali come le abbiamo viste nel paragrafo precedente.

Possono esservi più clausole ELSIF ... seguito 2 o non esservi. La clausola ELSE .... seguito 3 può mancare.

Il tutto è eseguito come segue:

— Se la condizione 1 è vera, si effettua il seguito delle istruzioni 1 e poi si passa al 4.

— Se la condizione 1 non è vera, e se la condizione 2 lo è, si esegue il seguito di istruzioni 2 e poi si passa al 4.

— Se ci sono diverse clausole ELSIF vengono esaminate nell'ordine (se la condizione 1 è falsa) e viene eseguito il seguito 2 corrispondente alla prima condizione 2 trovata vera. Si passa poi al seguito 4 senza esaminare le altre condizioni 2 (anche se alcune possono essere vere).

— Se nessuna condizione tra la 1 e la 2 è vera si esegue il seguito 3 (se ha un ELSE) poi si passa al seguito 4. Da ultimo, ELSE è identico a ELSIF TRUE.

Vediamo che si usano volentieri nei casi particolari le forme semplici IF .... THEN ... END IF e IF .... THEN ... ELSE ... END IF;

END IF; evita la difficoltà dei punti e virgola di Pascal che bisognava esaminare con cura. D'altra parte, è possibile in ogni clausola mettere dei seguiti di istruzioni. Non siamo obbligati a creare un blocco con BEGIN .... END come in Pascal (i blocchi servono ad altro scopo in ADA).

Si possono avere degli IF annidati? Notiamo che la possibilità di clausole ELSIF rende ciò meno utile. Tuttavia, le regole del linguaggio non lo impediscono.

Si arriva tuttavia a delle strutture che non sono sempre molto facili da comprendere. Tutto il problema consiste nel sapere a quale IF si riferisce un ELSIF o un IF; la regola è evidente. Un ELSIF o un ELSE si riferiscono all'ultimo IF incontrato per il quale non si è ancora trovato un END IF.

In:

```
IF A THEN
  1—;
   IF B THEN
    2—;
   ELSIF C THEN
    3—;
   END IF;
  4—;
ELSIF D THEN ...
```

le istruzioni da 1 a 4 formano la clausola THEN del primo IF. Si verifica che c'è un IF all'interno. L'ELSIF C si riferisce a IF B. ELSIF D si riferisce a IF A poichè nel momento in cui lo si incontra, l'IF B è stato chiuso dal suo END IF.

ESERCIZIO 2.8: A quale condizione è eseguita l'istruzione 3 nell'esempio qui sopra? Quale istruzione è effettuata in seguito?

ESERCIZIO 2.9: Date due versioni di un pezzo di programma che risolve l'equazione di primo grado AX+B=0

BLOCCHI

Si può raggruppare una sequenza d'istruzioni per formare un blocco compreso fra BEGIN e END;

Un blocco può ricevere un nome (che obbedisce alle regole abituali degli identificatori). Il nome è dato nell'intestazione, seguito da due punti. Deve essere ricordato nell'END.

Ciò che costituisce una novità, è che un blocco può cominciare con una parte dichiarativa (introdotta da DECLARE).

Le variabili dichiarate nel blocco sono **locali** al blocco (inaccessibili dall'esterno). Se un identificatore è ridichiarato in un blocco, è creata una nuova versione dell'identificatore, che **nasconde** l'interpretazione esterna

Esempio: blocco di scambio fra A e B:

```
SCAMBIO:
 DECLARE
         TEMP: tipo voluto;
 BEGIN
         TEMP:=A;
         B:=B;
         B:=TEMP;
 END SCAMBIO;
```

TEMP non è accessibile dall'esterno di SCAMBIO. Se all'esterno c'è una variabile TEMP, essa è diversa da quella all'interno del blocco.

CICLI

I cicli sono tutti introdotti dalla parola-chiave LOOP, ma ve ne sono di più tipi.

Ciclo indefinito

La struttura:

```
LOOP
  istruzioni;
END LOOP;
```

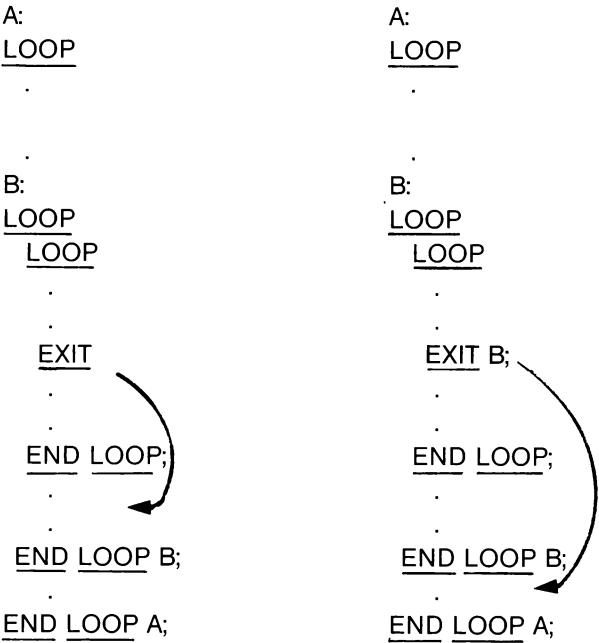
fa ripetere le istruzioni indefinitamente. È un principio da evitare, bisogna proprio che vi sia una condizione d'arresto.

La condizione d'arresto potrebbe anche essere segnata all'interno del ciclo da un IF condizione THEN GOTO ... ma questa è contraria alle prescrizioni della programmazione strutturata. (A dire il vero, tutte le uscite dai cicli senza che l'argomento abbia assunto tutti i valori possibili sono criticate dalla programmazione strutturata).

EXIT

Un modo migliore di uscire dal ciclo è di usare l'istruzione EXIT che rinvia fino all'ultimo END LOOP;

Con un EXIT situato all'interno del più interno di una serie di cicli annidati si esce unicamente dal ciclo più interno. Ma i cicli possono ricevere un nome (come i blocchi) e l'EXIT può designare il nome del ciclo da cui si esce (si esce nello stesso tempo dai cicli contenuti in esso).

Esempio:

Quando il ciclo ha un nome, questo deve essere richiamato nell'END LOOP.

Certamente, EXIT può trovarsi in una istruzione nella forma:

```
IF condizione THEN EXIT;
```

Ciò può essere espresso sotto la forma:

```
EXIT WHEN condizione;
```

EXIT può figurare anche in cicli a iterazioni esplicite che ora vedremo.

Ciclo WHILE:

Quando EXIT WHEN figura alla fine di un ciclo indefinito:

```
A : LOOP
    .


    .
  EXIT A WHEN condizione;
  END LOOP A;
```

si ottiene un REPEAT UNTIL della programmazione strutturata.

Esiste anche WHILE e si scrive:

```
WHILE condizione LOOP
    .


    .
END LOOP;
```

ESERCIZIO 2.10: si dice che la successione

$$u_0 = 1$$

$$u_n = \frac{1}{2}\left(u_n - 1 + \frac{a}{u_n - 1}\right)$$

converge a $\sqrt{a}$. Calcolare $\sqrt{a}$ con la precisione relativa di un millesimo.

Cicli con Indice

Come abbiamo appena visto, il ciclo WHILE permette molto comodamente d'iterare fino a che sopraggiunge una certa condizione. Per esempio, si percorrerà un file finchè si arriva alla fine del file.

Negli altri casi si può volere fare un'azione per un insieme di elementi, ad esempio per tutti gli elementi di una matrice. Perciò ci occorre far variare un indice I su tutti i valori possibili. Avendo:

A:ARRAY (1...N) OF FLOAT;

se si volesse calcolare la somma degli A, si scriverebbe:

```
I=1; S:=0.0;
WHILE  I<N  LOOP
     S:=S+A(I);
     I:=I+1;
END   LOOP ;
```

Ma si evita la gestione esplicita della variabile I scrivendo:

```
S:=0.0;
FOR  I  IN  1..N  LOOP
   S:=S+A(I);
END   LOOP ;
```

Si evita anche la dichiarazione della variabile I. Questa è implicitamente dichiarata arrivando nel FOR come variabile locale del sotto-programma nel quale ci si trova.

Questo risolve uno dei problemi più preoccupanti di altri linguaggi dalle variabili troppo globali. La variabile I non può essere modificata all'interno del ciclo.

Il FOR poteva anche essere scritto:

FOR I IN A 'RANGE LOOP

che è estremamente chiaro da comprendere. Per I nell'intervallo degli indici della matrice A, fare... In questo modo, a può essere indicizzato, ad esempio da —5 a +5, o essere indicizzato da qualsiasi tipo numerico scalare.

Ricordiamo che si può specificare qualsiasi intervallo discreto:

FOR I IN FRUTTO LOOP

o

FOR I IN MELA..ARANCIA LOOP

L'intervallo può essere percorso in decrescendo (equivalente al downto di Pascal), scrivendo:

FOR I IN REVERSE ...

Per esempio, per calcolare il fattoriale N, si può scrivere:

```
F:=1.0;
FOR  I  IN   REVERSE  1..N  LOOP
      F:=F*I;
END   LOOP ;
```

ESERCIZIO 2.11: Dato un vettore A(1...N), si vuol sapere se il valore B è presente in questo vettore e in quale posizione K. Altrimenti, si risponde K=0 oppure K=N+1. Dare differenti soluzioni per evidenziare le possibilità offerte dal linguaggio.

COSTRUTTO CASE

Questa struttura consente di inserire dei test a più uscite. Essa ha una clausola di raggruppamento delle ipotesi non prese in considerazione, analoga alla clausola OTHERWISE che non tutti i Pascal hanno.

Essa si presenta sotto la forma:

```
CASE  espressione  IS
      WHEN  valore 1   valore 1'  =>  istruzione 1;
      WHEN  valore 2                  istruzione 2;
      WHEN   OTHERS               =>    istruzione 3;
END   CASE ;
```

I differenti valori devono essere delle costanti appartenenti allo stesso tipo dell'espressione. Certamente, l'espressione non potrà prendere che uno dei diversi valori previsti, e a questo punto, è l'istruzione (o la sequenza di istruzioni) data in proposito ad essere effettuata (poi si passerà oltre END CASE).

L'esempio dimostra che si possono avere più valori per un'istruzione. Dei valori isolati sono separati da | ; si può anche specificare tutto un intervallo di valori V1...V2.

Quando nessuno dei valori previsti compare, si effettua l'istruzione che corrisponde a OTHERS. La clausola WHEN OTHERS può non comparire,

ma se c'è, significa che tutti i valori che l'espressione può assumere, tenuto conto del suo tipo, sono rappresentati:

```
TYPE  COLORE  IS  (BLU,BIANCO,ROSSO);
 X:COLORE;
-----

CASE  X  IS
    WHEN  BLU = >...;
    WHEN  ROSSO= > ...;
END   CASE ;
```

è illegale perché BIANCO non è previsto. Manca sia un WHEN BIANCO, che un WHEN OTHERS.

Si può, per fare un'ipotesi, mettere un'istruzione vuota (NULL;) se per questa ipotesi, non c'è nessuna azione da effettuare.

... WHEN OTHERS =>NULL;

ESERCIZIO 2.13: In una record riguardante un cliente, si è letto un carattere (LC:CHARACTER) che determina lo sconto da praticare ad un cliente secondo la regola:

'A' sconto 10% (dunque DA-PAGARE:=0.9*IMPORTO)
'B' sconto 5% se l'importo supera 100.000 lire, altrimenti nulla
'C' sconto 10%
'D' sconto 5% (in ogni caso)
'E' nessuno sconto

CAPITOLO 3

TIPI DI DATI

Gli oggetti semplici che possono essere manipolati in ADA sono le variabili e le costanti. Qui *"semplice"* esclude gli oggetti come le procedure, i package, i task...

La dichiarazione di un oggetto si esprime nella forma:

nome dell'oggetto :[CONSTANT] indicazione di tipo [:=valore iniziale];

Come indicano le parentesi quadre, CONSTANT può essere specificato oppure no, come l'inizializzazione. Allorché la parola CONSTANT è specificata, l'oggetto è una costante. Nella maggior parte dei casi, è specificato un valore iniziale e nessun altro valore potrà essere dato all'oggetto nel corso del programma, poichè una costante non può essere oggetto di un'assegnazione. Allorchè non è specificato alcun valore, si tratta di una *costante differita* appartenente ad un tipo privato (vedere pp. 60 e 82).

Se l'oggetto non è una costante, costituisce una *variabile*. Il valore specificato non è che un valore iniziale suscettibile d'essere cambiato ad ogni istante nel programma da un'istruzione di assegnazione.

Il valore può essere specificato sotto forma di una literal o sotto forma di un'espressione aritmetica che deve poter essere interamente calcolata secondo l'elaborazione della definizione.

Tipi anonimi

L'indicazione di tipo può essere formata sia dal nome di un tipo, prece-

dentemente definito con una dichiarazione del tipo secondo le forme:

TYPE nome del tipo IS definizione del tipo;

oppure

SUBTYPE nome di tipo IS ....;

oppure può essere direttamente formata da una definizione di tipo. Si ha allora un tipo anonimo.

Esempi:

```
X:REALE;—— variabile del tipo REALE
PI: CONSTANT FLOAT:=3.14159;—— la costante PI
V: ARRAY (1..N) OF FLOAT;——V vettore di un tipo anonimo
                          ——matrice(1..N)
```

se si rimpiazza la dichiarazione di V con:

```
TYPE VETTORE IS  ARRAY (1..N) OF FLOAT;
V:VETTORE;
```

si ha lo stesso risultato per V, ma il tipo non è più anonimo.

L'uso di un tipo anonimo è utile soltanto nel caso in cui ci sia un solo oggetto di questo tipo da dichiarare e l'oggetto è utilizzato localmente. Si evita così di dedicargli un identificatore.

Altrimenti, ad utilizzare i tipi anonimi si hanno soltanto inconvenienti e *noi li sconsigliamo formalmente*.

Infatti, se si ha:

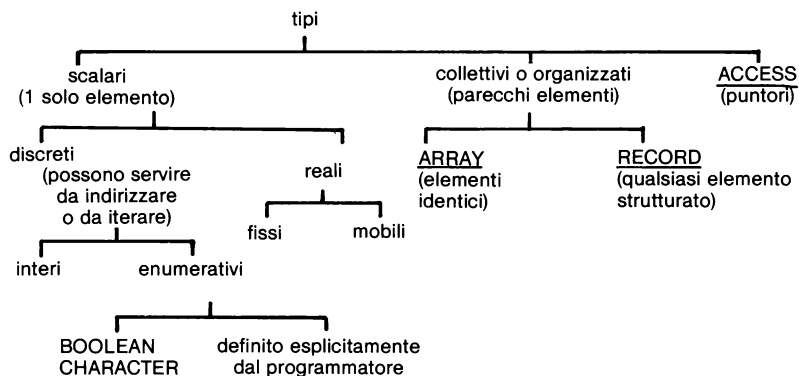
A,B: definizione di tipo;
C: esattamente la stessa definizione;

A e B saranno considerati come dello stesso tipo. C sarà considerato di un tipo differente, pur avendo le stesse proprietà.

Di conseguenza, se A:=B; è legale, C:=B; sarà illegale (assegnazione con tipi differenti!).

Allo stesso modo, oggetti di tipo anonimo non possono servire da argomento per una procedura. Non saranno mai il tipo ideale.

CLASSIFICAZIONE DEI TIPI

Si possono classificare i tipi secondo lo schema qui sotto:

Nei tipi scalari, il dato è monolitico, laddove i tipi collettivi definiscono degli insiemi di dati, dello stesso genere per una matrice, di genere diverso per un RECORD.

Così, un ARRAY(1..10) OF INTEGER è formato da 10 elementi che sono tutti interi, mentre:

```
RECORD
   NOME: ARRAY (1..10) OF  CHARACTER;
   MATRICOLA:INTEGER;
   SITFAM:(SPOSATO,CELIBE,DIVORZIATO, VEDOVO);
END  RECORD ;
```

è formato dall'unione di una stringa di dieci caratteri, di un intero e di una scelta fra le quattro possibilità enumerate.

I tipi ACCESS sono i puntatori che consentono di gestire dinamicamente la memoria.

Fra i tipi scalari, si distinguono i tipi reali e quelli discreti.

I tipi discreti sono formati da un insieme ordinato di valori, numerabile (l'insieme dei valori reali è anch'esso numerabile nel senso della teoria degli insiemi, poiché gli elaboratori non possono che darne un'approssimazione discreta. Tralasciamo questo problema. I reali vogliono essere un'approssimazione di un insieme che, idealmente, non è numerabile).

•I tipi discreti possono servire da indici per gli array o da indici di ciclo in un'iterazione.

Fra i tipi discreti, si trovano **gli interi e i tipi enumerativi.** Un tipo enumerativo è definito dall'enumerazione di un numero finito di valori possibili. Ci sono due tipi enumerativi predefiniti:

```
TYPE BOOLEAN IS (FALSE,TRUE);
TYPE CHARACTER IS (NUL,SOH,...'A','B',....);
```

Altrimenti, è il programmatore che definisce la lista degli identificatori delle costanti astratte che formeranno il tipo.

Esempio:

```
TYPE DIREZIONE IS (AVANTI,INDIETRO,SINISTRA,DESTRA);
```

questa enumerazione definisce l'ordine dei valori:

```
AVANTI<INDIETRO è TRUE.
```

I tipi numerici sono gli interi e i reali. Si possono avere delle conversioni dall'uno all'altro.

Qualunque sia il tipo, vi sono sempre due cose che possono essere fatte fra due elementi di uno stesso tipo.

Si tratta dell'**assegnazione** A:=B e il **test di uguaglianza** A=B (e, di conseguenza, A/=B).

Per impedire queste due cose, ed arrivare così all'ultimo dei tipi privati, occorre una menzione speciale: LIMITED (cfr. pp. 81 e 84).

Per un tipo collettivo, l'assegnazione è un'assegnazione multipla, elemento corrispondente ad elemento corrispondente.

Le altre operazioni, sono definite oppure no a seconda che abbiano o meno un senso in funzione della natura dei dati descritti dal tipo. Per esempio, l'operazione di addizione per un intervallo di interi ha ogni motivo di esistere. D'altra parte, la moltiplicazione di colori non avrà probabilmente senso.

Questa definizione di operazioni per nuovi tipi si chiama "**sovraccarico**". A seconda del modo in cui il nuovo tipo è definito, questo sovraccarico può essere automatico (è il caso per esempio dei tipi derivati), o può aver bisogno di una definizione esplicita.

TIPI DERIVATI E SOTTO-TIPI

Ci sono due modi di ottenere un tipo partendo da un altro tipo (predefinito nel linguaggio, o precedentemente definito dal programmatore):

Si può formare un tipo derivato mediante:

TYPE nome IS NEW nome del tipo genitore {vincolo};

Il tipo derivato "eredita" tutte le costanti e tutte le operazioni del tipo genitore. Forma pertanto un tipo distinto e può servire come esempio per distinguere degli insiemi logicamente differenti.

Esempio:

con:

```
TYPE  COLORE  IS (VERDE, GIALLO ROSSO);
TYPE  SEMAFORO  IS  NEW  COLORE;
PASSAGGIO: SEMAFORO:=VERDE;
TINTA:COLORE;
```

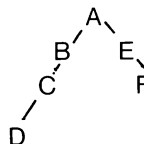
TINTA=PASSAGGIO; sarà illegale (tipi differenti) e ciò è normale.
Si può passare da tipi genitori a derivati molto facilmente: è sufficiente dire il nome del tipo (oggetto). Per esempio, l'assegnazione illegale qui sopra avrebbe potuto scriversi legalmente:

TINTA:=COLORE(PASSAGGIO);

Si possono avere dei canali di derivazione:

```
TYPE  A...;
TYPE  B  IS  NEW  A;
TYPE  C  IS  NEW  B;
TYPE  D  IS  NEW  C;
TYPE  E  IS  NEW  A;
TYPE  F  IS  NEW  E;
```

corrispondono allo schema:

```
        A
       / \
      B   E
     /     \
    C       F
   /
  D
```

Con una simile struttura, le conversioni non possono avere luogo che fra tipi adiacenti, per esempio, da A a B, da B a C ....

ESERCIZIO 3.1: coi tipi qui sopra e:

BB:B;
FF:F;
BB:=B(FF); è illegale. Come fare?

Nel nostro esempio dei colori, un FOR I IN VERDE...ROSSO sarebbe ambiguo poiché i simboli sono sovraccaricati su due tipi. L'ambiguità può essere tolta qualificando i simboli: FOR I IN COLORE'(VERDE)...COLORE'(ROSSO) LOOP

Si può formare un sottotipo con:

SUBTYPE nome IS nome di tipo genitore (vincolo);

Un sotto-tipo non introduce un tipo nuovo. In particolare, quando non c'è un vincolo, una dichiarazione di sotto tipo interviene a cambiare il nome del genitore.

Esempio:

```
TYPE COLORE IS
(ROSSO,ARANCIONE,GIALLO,VERDE,BLU,VIOLA,INDACO);
SUBTYPE ARCOBALENO IS COLORI;
```

Ne risulta che elementi di un tipo e dei suoi sottotipi possono essere mescolati liberamente nelle espressioni aritmetiche, nel rispetto dei vincoli.

VINCOLI

Al contrario dei primi esempi qui sopra, più sovente la definizione di un tipo è accompagnata da un vincolo che va a restringere l'insieme dei valori autorizzati per il tipo.

Il vincolo si definisce in rapporto al tipo genitore da cui si parte, potendo questo tipo genitore essere uno dei tipi predefiniti del linguaggio che hanno essi stessi una determinata gamma di valori ammissibili.

I vincoli sono di genere diverso secondo il tipo al quale si applicano.

Tipi discreti:

Nei tipi discreti, si trovano dei vincoli d'intervallo:

TYPE MILLE IS NEW INTEGER RANGE 1...1000;
TYPE INDICE IS NEW INTEGER RANGE 1...N;

(I limiti non possono essere conosciuti che durante l'esecuzione)

TYPE GIORNO IS
(LUNEDI,MARTEDI,MERCOLEDI,GIOVEDI,VENERDI,SABATO,
DOMENICA);
SUBTYPE GIORNO - FERIALE IS GIORNO RANGE
LUNEDI...SABATO;

Una dichiarazione di tipo può riferirsi ad un tipo genitore anonimo:

TYPE MILLE IS RANGE 1...1000; non si riferisce più a
INTEGER

Una dichiarazione di sottotipo deve sempre far riferimento ad un tipo nominato:

SUBTYPE MILLE IS RANGE 1...1000; sarebbe illegale.

La forma corretta è:

SUBTYPE MILLE IS INTEGER RANGE 1...1000;

I tipi interi possono riferirsi a tre tipi predefiniti: INTEGER, LONG-INTEGER e SHORT-INTEGER.

Tipi reali:

I tipi reali hanno dei vincoli d'intervallo e dei vincoli di precisione, presentati sotto la forma distanza assoluta fra numeri per i numeri a virgola fissa, e sotto la forma numero di cifre significative per i numeri a virgola mobile:

Esempio:

TYPE REALE IS NEW FLOAT DIGITS 6 RANGE —1.0E+10...1.0E+10;

definisce un tipo reale di almeno sei cifre significative, di valore compreso fra -10^{10} e 10^{10}; si tratta di un tipo a virgola mobile.

TYPE FIX IS NEW FLOAT DELTA 0.001 RANGE 0.0..5.0;
definisce un tipo in virgola fissa che ricopre l'intervallo 0-5 a passi di 1/1000.

Il vincolo d'intervallo può mancare: l'intervallo va allora dal numero più piccolo al più grande rappresentabile dal computer.

Quando si definisce un tipo reale, si definisce un insieme di **numeri modello** che sono le rappresentazioni binarie possibili con un numero di bit sufficiente a soddisfare il vincolo di precisione.

Per un tipo in virgola mobile, definito da DIGITS D, si prenderanno B bit con B=intero immediatamente superiore a D log 10/log 2= D/0,30103.

I numeri modello sono allora 0 e tutti i numeri della forma:
± mantissa * 2 * * esponente.

— **mantissa** è della forma 0,XXX...X: cioè uno dei numeri rappresentabili esattamente in binario con B bit e compresi fra 0,5 e 1.
— **esponente** è un intero compreso fra —4*B e +4*B.

Quando ci si riferirà ad un numero nel tipo, per esempio una literal, questo numero sarà arrotondato al modello più vicino. Non disterà mai dal vero numero che si voleva rappresentare di uno scarto superiore alla precisione richiesta.

Per un tipo fisso, definito da DELTA X, si definisce un **delta-binario** o **delta effettivo** che è sovente la potenza di 2 immediatamente inferiore a X.

I numeri modello sono allora della forma K*delta- binario dove K appartiene all'insieme degli interi compresi fra -2^{N+1} e 2^{N-1}. N è scelto affinché i limiti richiesti siano compresi nell'intervallo definito dai numeri modello.

ESERCIZIO 3.2: Si definisce:

TYPE FF1 IS NEW FLOAT DIGITS 1;

Descrivere i numeri modello. Quale sarà l'approssimazione di 0.3? Stessa domanda per DIGITS 3.

ESERCIZIO 3.3: Che cos'è che non va in

TYPE FF2 IS NEW FLOAT DIGITS 2 RANGE 0.0..1.0E20;

ESERCIZIO 3.4: Descrivere i numeri modello di:

TYPE FF1 IS NEW FLOAT DELTA 0.01 RANGE 0.0...10.0

Tipi reali predefiniti

C'è sempre un tipo predefinito FLOAT le cui caratteristiche sono legate al calcolatore considerato. Un compilatore ADA può anche predefinire due altri tipi reali: LONG-FLOAT e SHORT-FLOAT che saranno rispettivamente più preciso e meno preciso di FLOAT.

La possibilità per il programmatore di esigere la precisione che gli conviene è un eccellente fattore di portabilità.

Si raccomanda di definire per esempio:

TYPE REALE IS NEW FLOAT DIGITS 10;

Se si deve ora passare su di una macchina dove il tipo FLOAT non può assicurare DIGITS 10, basterà scrivere:

TYPE REALE IS NEW LONG-FLOAT DIGITS 10;

e sarà l'unica istruzione del programma da riscrivere!

In realtà sarebbe stata sufficiente l'istruzione

TYPE REALE IS DIGITS 10;

che avrebbe indotto il compilatore ad avere scelto da solo il tipo di genitore che può soddisfare il vincolo.

ESERCIZIO 3.5: sarebbe la stessa cosa se REALE fosse un SUBTYPE?

Bisogna notare che DIGITS 10 è compreso sotto la forma di "almeno 10 cifre". Il sistema assicura una precisione almeno uguale alla precisione richiesta, può anche darne di più.

Dunque, se la precisione richiesta è sufficiente ad assicurare il buon funzionamento del vostro algoritmo, potete essere sicuri del risultato su qualsiasi computer.

ADA possiede perciò un eccellente strumento di portabilità per i problemi di analisi numerica. Certo, le considerazioni precedenti si applicano anche ai vincoli DELTA.

ATTRIBUTI

Gli attributi sono in un certo senso delle funzioni legate sia ai tipi, sia agli oggetti in un tipo dato. Queste funzioni rappresentano delle informazioni legate al sistema o alla rappresentazione nel sistema degli oggetti considerati: pochi linguaggi forniscono l'accesso a tali informazioni.

Per esempio, X'ADDRESS è l'indirizzo della prima parola di memoria riservata all'oggetto X. Questo attributo è definito per qualsiasi oggetto, variabile o sottoprogramma.

La forma di annotazione precedente **elemento nome dell'attributo** è usata per tutti gli attributi.

Per ogni tipo o sottotipo T, sono definiti:

T'BASE : per un sottotipo, è il tipo genitore; per un tipo, è il tipo stesso.

T'SIZE : dimensione massima di memoria (in bit) necessaria per un elemento del tipo T.

Per ogni tipo o sottotipo scalare T, si definisce:

T'FIRST: l'elemento minimo del tipo.

Esempio:
Se:

```
TYPE COLORE IS (BLU, BIANCO,ROSSO);
COLORE'FIRST è BLU.
```

T'LAST: elemento massimo del tipo
SYSTEM.MIN—INT e SYSTEM.MAX—INT sono il T'FIRST e il T'LAST del tipo intero predefinito che ha l'intervallo più grande.

T'IMAGE (X): rappresentazione della stringa di caratteri dell'oggetto X del tipo T. Per un oggetto numerico, è ciò che in BASIC si scriverebbe STR$(X). Per un tipo enumerativo, è il nome della costante astratta che corrisponde al valore:

se:
X:=BLU

COLORE'IMAGE (X) è la stringa di tre caratteri BLU. Per un tipo di carattere enumerativo, si ottiene una literal fra apostrofi.

T'VALUE(X): è il valore nel tipo T dell'oggetto che sarebbe rappresentato dalla stringa di caratteri X. È il reciproco di IMAGE.

Per ogni tipo o sottotipo discreto T, sono definiti:

T'POS(X) : posizione dell'oggetto X del tipo T nella sequenza T'FIRST.. T'LAST, sapendo che per un tipo intero T'POS(T'FIRST)=T'FIRST, e per un tipo numerativo, T'POS(T'FIRST)=0. In sostanza, è l'ORD di Pascal.

ESERCIZIO 3.6: Secondo l'esempio qui sopra, quanto vale COLORE'POS (BIANCO)?

T'VAL(P) : è il valore dell'oggetto di T dove POS è P.

T'PRED(X) : è il predecessore di X nel tipo T. Non è definito per X=T'FIRST.

T'SUCC(X) : è il successore di X nel tipo T. Occorre che sia X/=T'LAST.

ESERCIZIO 3.7: quale relazione intercorre fra:

T'POS(T'SUCC(X)) e T'POS(X)?

I principali attributi dei tipi reali in virgola fissa sono:

T'DELTA che è il DELTA quale lo si è specificato

T'ACTUAL-DELTA che è il delta-binario effettivamente utilizzato

T'BITS	che è il numero di bit necessari per rappresentare i numeri modello di T
T'LARGE	che è il più grande numero modello di T

I principali attributi dei tipi reali a virgola mobile sono:

T'DIGITS	che è il numero di cifre specificate nella dichiarazione del tipo
T'MANTISSA	che è il numero di bit nella rappresentazione della mantissa dei numeri modello, (è il B dell'esercizio 3.2)
T'EMAX	che è l'esponente più grande per i numeri modello, (l'esponente più piccolo è T'EMDICE)
T'SMALL	che è il più piccolo numero modello positivo (il minor numero che si possa formare al di fuori dello 0). È dell'ordine di 2**(-T'EMAX)
T'LARGE	che è il massimo numero modello
T'EPSILON	che è la differenza fra 1.0 (che è sempre un numero modello) e il minor numero modello >1.0
T'MACHINE-MANTISSA	che è il numero di bit della mantissa nella rappresentazione della macchina usata. È ≥ T'MANTISSA.

Per ogni tipo reale, si ha T'LAST≤T'LARGE
Vedremo gli attributi dei tipi collettivi nel paragrafo seguente.

OPERAZIONI

Le operazioni permesse su tutti i tipi sono, l'abbiamo visto, l'assegnazione e il test d'uguaglianza (e diseguaglianza).

Le operazioni permesse sui tipi numerativi sono, inoltre, le operazioni di confronto come X<Y. Il risultato è booleano e definito dall'ordinamento generato dall'enumerazione.

Per esempio, nel TYPE COLORE IS (BLU,BIANCO,ROSSO); BLU<ROSSO è TRUE.

Attenzione: Allorchè dei simboli sono sovraccaricati, non è permessa alcuna ambiguità.

Esempio:

Se si ha:

TYPE COLORE IS (BLU,BIANCO,ROSSO);

e

TYPE TINTA IS (BLU,VERDE,GIALLO,ARANCIONE,ROSSO);
BLU<BIANCO fa intervenire il BLU di COLORE;
BLU<VERDE fa intervenire il BLU di TINTA;

ma

BLU<ROSSO è ambiguo e dunque vietato. Bisogna scrivere COLORE'(BLU)<COLORE'(ROSSO) o TINTA'(BLU)...per distinguere.

Per il tipo booleano, si dispone inoltre delle operazioni logiche (NOT,AND ecc...).

Coi tipi interi, si dispone delle operazioni aritmetiche abituali. La divisione è la divisione intera. MOD fornisce il modulo mentre REM fornisce il resto.

Quando A e B sono entrambi positivi, o negativi, A REM B e A MOD B coincidono. Altrimenti, REM ha il segno di A, MOD ha il segno di B.

ESERCIZIO 3.8: completare la tabella:

A	B	A/B	A REM B	A MOD B
10	3			
9	3			
-10	3			
-9	3			
10	-3			
9	-3			
-10	-3			
-9	-3			

L'operazione di esponenziazione non è possibile che per A**B con A intero e B intero positivo o nullo.

Coi tipi reali, la divisione è fatta in aritmetica reale. Per i tipi a virgola fissa, sono definite le seguenti combinazioni:

A	B	A*B	A/B
fisso	intero	tipo di A	tipo di A
intero	fisso	tipo di B	no
fisso	fisso	fisso di precisione massimale : sara' convertito in seguito	fisso di precisione massimale

Per l'esponenziazione è permesso solo: virgola mobile ** intero. Se B è negativo, si ottiene A**B uguale a 1.0/A**ABS(B). A**0.0 vale 1.

TIPI MATRICE

La dichiarazione di un tipo matrice definisce gli indici che saranno ammessi e il tipo degli elementi. La matrice può essere vincolata (a determinate dimensioni) o no.

TYPE VETTORE IS ARRAY (INTEGER < > RANGE) OF FLOAT;

è *non vincolato.*

TYPE VETT IS ARRAY (1...10) OF FLOAT;

è *vincolato.*

Si può anche scrivere SUBTYPE VETT IS VETTORE (1...10);

Si possono avere delle matrici multidimensionali e qualsiati tipo discreto può servire da indice:

TYPE MATRIMONIO IS ARRAY (COLORE,COLORE) OF BOOLEAN;
V:MATRIMONIO;

Si potrà avere V(BLU,GIALLO):=TRUE; per dire che il blu si sposa bene col giallo.

Una matrice multidimensionale può essere definita sia come tale, sia come matrice di matrici.

ATTRIBUTI DELLE MATRICI

Data una matrice M,

M'FIRST(I) è il limite inferiore dell'indice I-esimo di M

M'LAST(I) è il limite superiore dell'indice I-esimo di M

M'LENGTH(I) è il numero di valori possibili per l'indice I-esimo di M (0 per una dimensione vuota)

M'RANGE(I) è il sottotipo definito dall'intervallo M'FIRST(I)...M'LAST(I).

Per l'indice 1 (o un indice unico), si omette (I). Si scrive per esempio: V'FIRST.

Attenzione: V'FIRST è l'indice del primo elemento; non è la prima componente:

V'FIRST=V'RANGE'FIRST

ESERCIZIO 3.9: se la matrice è definita da:

TYPE MAT-RECT IS ARRAY(1... 10,1...20) OF FLOAT;
M:MAT-RECT;

date gli attributi di M.

OPERAZIONI SULLE MATRICI

L'attribuzione e il test di uguaglianza sono effettuati un elemento per voita. C'è uguaglianza se tutti gli elementi corrispondenti sono eguali. Rivedere a pag. 26 gli aggregati che possono servire da valori in una assegnazione ad una matrice.

ESERCIZIO 3.10: assegnare il valore vettore-nullo al vettore V.

Per le matrici monodimensionali, è definita l'operazione di concatenazione ed è indicata con &: essa forma l'unione dei suoi due operandi. Se:

U,V,W : ARRAY (INTEGER RANGE < >) OF T;
W:=U+V;
W è come :
W'LENGHT=U' LENGHT + V'LENGHT
W(1..V'LENGHT) ≡ U
W(V'LENGHT + 1..W:LENGHT) ≡ V

Segmenti

Generalizziamo la precedente osservazione.

Se A e B appartengono a V'RANGE, V(A...B) è il vettore formato dalla sequenza delle componenti A,A+1,...B di V.

Le assegnazioni fra segmenti sono possibili:

V(1...5):—W(I...J); —— qui occorre che J=I+4

ma per dei segmenti presi nella stessa matrice bisogna che non vi sia sovrapposizione.

RICERCA IN UNA MATRICE

Data una matrice monodimensionale TAB, K è una costante dello stesso tipo degli elementi di TAB. Si vuole stampare la posizione del primo elemento di TAB uguale a K se esiste, o un messaggio se non esiste.

Si introduce il booleano TROVATO che sarà TRUE se esiste un elemento = K.

Con le nozioni viste, si può proporre il programma:

```
TROVA:=FALSE;
FOR  I  IN  TAB'RANGE  LOOP
       IF  TAB(I)=K  THEN
           PUT(I);-- si stampa la posizione
           TROVATO:=TRUE;
            EXIT ;
            END   IF ;
      END   LOOP ;
IF     NOT  TROVATO  THEN
         PUT("NON TROVATO");
          END   IF ;
```

ESERCIZIO 3.11: Scrivere le dichiarazioni corrispondenti a questo programma.

ESERCIZIO 3.12: Se si è supposta la matrice ordinata (in ordine crescente), si può fare una ricerca dicotomica nella quale si divide ogni volta per due l'intervallo di ricerca. Ci si colloca ogni volta nel centro I dell'intervallo. Se TAB(I)=K, è trovato. Se TAB(I)> K, significa che bisogna continuare a cercare nella metà inferiore.

Si vede che le matrici si manipolano come nei linguaggi classici, con delle caratteristiche supplementari apportate dagli attributi.

Consigliamo al lettore di allenarsi a scrivere un prodotto scalare, un prodotto di matrici, ecc..

STRINGHE DI CARATTERI

Un caso particolare di matrice monodimensionale è il tipo predefinito STRING:

TYPE STRING IS ARRAY (NATURAL RANGE < >) OF CHARACTER;

CHARACTER è il tipo predefinito formato dall'enumerazione dei 128 caratteri ASCII.

NATURAL è il sottotipo predefinito da:

SUBTYPE NATURAL IS INTEGER RANGE 1...INTEGER'LAST;

L'indice in una stringa inizia sempre da 1. Una stringa è definita abbreviando la sua lunghezza massima:

STRINGA:STRING(1...80)

Un vincolo di questo tipo si avvicina molto alla specificazione di un parametro: è premura costante di ADA consentire tali parametrizzazioni.

La lunghezza massima ed effettiva può essere stabilita all'atto dell'inizializzazione:

STRINGA:STRING:="BUONGIORNO"; —— lunghezza mass.=10

Le costanti stringa di caratteri sono sia degli aggregati ('B','U','O','N','G','I','O','R','N','O') o (1| 2='X',OTHERS=>' '), sia delle stringhe fra virgolette: "BUONGIORNO" o "XX".

In ADA sono offerte la maggior parte delle possibilità classiche di trattamento delle stringhe di caratteri:

attribuzione:	già esaminata.
concatenazione:	è un caso particolare dell'operatore & che agisce sulle matrici unidimensionali.

estrazione, soppressione: è sufficiente fare intervenire un segmento. X(5..7) è la sottostringa formata dai caratteri 5,6,7 di X. (in BASIC, si scriverebbe MID$ (X$,5,3)).

confronti: oltre ai test di uguaglianza, in ADA sono definiti tutti gli operatori di confronto e si riferiscono all'ordine lessicografico.

Le funzioni LEN, VAL,...del BASIC sono esprimibili in ADA con l'aiuto degli attributi LENGTH, IMAGE, ecc... per esempio, ASC(A$) si scriverebbe CHARACTER'POS(A).

ESERCIZIO 3.13: Data una stringa X dal numero di caratteri imprecisato, trasformarla in una stringa Y formata da K caratteri, se K<X'LENGTH, coi K caratteri più a destra di X (RIGHT$ in BASIC) e se K>X 'LENGTH, Y è formata da X col numero di spazi a sinistra voluti perché Y abbia K caratteri (quadratura).

ESERCIZIO 3.14: Dire se la sottostringa X è presente nella stringa Y. Se sì, dare il posto nella Y del primo carattere dove comincia la coincidenza. Per SI nel "BUONGIORNO SIGNORA", è 12.

MATRICI COMPATTE

ADA non possiede la clausola PACKED di Pascal che specifica che una matrice deve essere collocata in memoria in modo compatto. Si può chiederlo — ma il compilatore non lo fa a meno che sia costruito per questo scopo — con il PRAGMA:

PRAGMA PACK (nome del tipo matrice o RECORD);

TIPI RECORD

Questi tipi consentono di creare un oggetto collettivo con la unione di componenti di diversa natura.

Sono molto simili ai RECORD di PASCAL e svolgono lo stesso ruolo delle strutture di PL/1 o COBOL.

Di tali dati eterogenei sono pieni i record dei file. Per esempio, in un file clienti, si avrà per ogni cliente:

— numero, nome, indirizzo, codice postale, il rappresentante che si occupa di lui, codice di avviamento postale, il giro di affari realizzato con lui.

Si definirà un tipo RECORD (RECORD vuole dire esattamente registrazione) sotto la forma:

```
TYPE  CLIENTE  IS
    RECORD
       NUMERO:INTEGER;
    NOME:STRING(1..10);
    INDIRIZZO:STRING(1..30);
    CPCITTA':STRING(1..20);
    NUMREP:INTEGER;
    CONSEGNA:BOOLEANO;
    CIFRAFF:FLOAT;
 END    RECORD  ;
```

Si potrà allora definire il cliente CLI con:

CLI:CLIENTE;

Un elemento di CLI sarà designato con un nome qualificato come CLI.INDIRIZZO:

CLI.NOME:="MARTINELLI"
IF CLI.CONSEGNA THEN...

ESERCIZIO 3.15: una linea di istruzioni comprende:

l'etichetta: 20 caratteri
prezzo unitario: reale
numero: intero
ammontare: reale

Dichiarate il tipo adeguato.

L'inizializzazione si può fare con aggregati. Per la linea di istruzioni dell'esercizio 3.15, si potrebbe avere:

```
LINEA:LICO;
LINEA:=("SOSTEGNI BASE",10.0,15,150.0)
```

oppure

```
LINEA:=(NUMERO=> 15,COMPILATO=>"SOSTEGNI BASE",
    PREZZOU => 10.0,IMPORTO=> 150.0)
```

o ancora

```
LINEA:LICO:=(....... );
```

Certi elementi possono essere dichiarati come costanti; una volta ricevuto un valore, non potranno più essere modificati.

Se l'inizializzazione è specificata nella dchiarazione del tipo, tutti gli oggetti del tipo ammettono questo valore per default (per le componenti per default).

Esempio:

Scrivendo:

```
TYPE  COMPLEX  IS   RECORD
      RE:FLOAT;
      IM:FLOAT:=0.0;
      END   RECORD  ;
```

tutti i numeri complessi saranno inizializzati come reali (parte immaginaria nulla).

RECORD CON VARIANTI

I differenti record di una stessa serie possono non avere esattamente la stessa struttura.

Prendiamo l'esempio degli impiegati di un'azienda. Il record conterrà, certamente, il nome e la data di nascita. In seguito, un dato indicherà la situazione di famiglia: 'C' celibe, 'S' sposato, 'D' divorziato, 'V' vedovo.

Il seguito della struttura dipende da questo elemento, che chiamiamo **discriminante.**

Nel caso celibe, non c'è più niente. Nel caso "sposato", ci sarà il nome del congiunto e la data di matrimonio. Nel caso "divorziato" o vedovo, c'è anche la data del divorzio o della vedovanza.

In ADA, il discriminante costituirà un parametro del RECORD. Si avrà per esempio:

```
TYPE  NNN  IS  STRING (1..10);
TYPE  FAM  IS  ('C','S','D','V');
TYPE  DATA  IS  RECORD
           GR:INTEGER  RANGE  1..31;
           MS:INTEGER  RANGE  1..12;
           AN:INTEGER  RANGE  0..99;
          END  RECORD ;

TYPE  IMPIEGATO(SITFAM:FAM) IS
       RECORD
          NOME:NNN;
          COGNOME:NNN;
          DTNAS:  CONSTANT  DATA;
           CASE  SITFAM  IS
              WHEN  'C' =>  NULL ;
              WHEN  'S' =>NOME CONGIUNTO:NNN;
                          COGNOME CONS  : NNN;
                          DATA MATR     : DATA;
             WHEN  'D'|'V' => DTVD   :DATA;
        END  CASE ;
    END  RECORD ;
```

Si vede che la data di nascita è stata specificata come costante: essa non ha infatti nessuna possibilità di variare nel corso dell'elaborazione. È una costante differita: sarà assegnata dalla prima attribuzione e non potrà più essere cambiata. Questa possibilità è stata sfortunatamente soppressa nell'ADA riveduto dove bisognerebbe scrivere DTNASC:DATA;

Si potrà scrivere, per esempio:

IMP:IMPIEGATO(C);
IMP:=("C","MARTINELLI","GIANNI",(15,8,40));

si può usare un aggregato chiamato:

IMP:=(SITFAM=>"C", NOME =>"MARTINELLI"....);

si possono assegnare le componenti:

IMP.NOME:="MARTINELLI"

ma qui non possiamo attribuire IMP.DTNASC, né IMP.SITFAM perché il

discriminante non può essere assegnato che all'atto dell'assegnazione di ogni RECORD.

Alcune componenti possono ricevere un valore iniziale che svolgerà il ruolo di valore per default. Questo può essere anche il caso del discriminante; se la maggioranza degli impiegati è sposata, è sensato scrivere:

TYPE IMPIEGATO(SITFAM:FAM:=S) IS

A questo punto, una dichiarazione come IMP:IMPIEGATO; sarà autorizzata e IMP sarà sposato anch'esso.

VINCOLI

I vincoli che possono essere applicati ai tipi RECORD sono dei valori imposti ai discriminanti.

Si potrebbe definire:

SUBTYPE IMP-CELIBE IS IMPIEGATO("C");

o procedere col tipo anonimo:

IMP:IMPIEGATO(SITFAM => "C");

Un'altra specie di discriminante fa intervenire come componente del record delle matrici di dimensione variabile, poiché questa dimensione costituisce il discriminante:

```
TYPE  CATENA(LUNGO:INTEGER  RANGE  0..MASS)  IS
     RECORD
        ENTRATA:INTEGER  RANGE  0..MASS:=0;
        USCITA :INTEGER  RANGE  0..MASS:=0;
        CONTENUTO: ARRAY  (1..LUNGO)  OF  tipo;
     END   RECORD ;
```

e si potrà avere:

BUFCIRC:STRINGA(100);

Quando un oggetto ha un tipo RECORD che ha ricevuto un tale vincolo, il suo attibuto booleano X'CONSTRAINED ha il valore TRUE.

Così:

```
BUFCIRC'CONSTRAINED è TRUE
```

mentre per:

```
BUF:STRINGA;
BUF'CONSTRAINED è FALSE.
```

ACCESSO ALLE COMPONENTI

Ben inteso, si può accedere alle componenti soltanto se esistono tenuto conto del valore del discriminante.

ADA include tutti i test possibili e attiva l'eccezione CONSTRAINT-ERROR se occorre.

La cosa migliore è che mettiate voi i test che sono necessari.

Per esempio:

```
IF IMP.SITFAM= 'S'  THEN
        PUT(IMP.DATAMATR);
```

Si usa sovente un CASE parallelamente al CASE del RECORD.

ESERCIZIO 3.16: Stampate l'anno del divorzio di IMP

Se X è una componente di RECORD, l'attributo X'POSITION è la distanza in parole fra l'inizio del RECORD e la componente X. In seguito al livello del bit, si definisce X'FIRST-BIT e X'LAST-BIT.

TIPI INSIEME

ADA non ha i tipi SET di Pascal, che, comunque, soffrivano di alcuni limiti.

In ADA, si procede per matrici booleane. Le operazioni d'insieme possono essere simulate dalle operazioni logiche (OR:unione; AND:intersezione; <=:inclusione).

Segnaliamo che l'appartenenza ad un insieme costituito da un intervallo si inizia con IN.

A titolo d'esempio, scriveremo una procedura che stampa i numeri primi da 2 a N. Questo ci permetterà di sfuggita d'iniziare a vedere come si manipolano le procedure.

Il metodo del crivello di Eratostene consiste nel formare una matrice dei numeri da 2 ad un numero voluto N; ad ogni iterazione si prende il numero più piccolo che resta nella matrice (la prima volta sarà 2). Lo si considera come primo, poi lo si elimina coi suoi multipli. Qui, il crivello sarà formato da una matrice di booleani indicizzata su 2...N. CRIB(K)=TRUE vorrà dire "ancora nella matrice"; CRIB(K)=FALSE vorrà dire "eliminato".

```
PROCEDURE  ERA  IS
     TYPE  NB  IS   NEW  INTEGER  RANGE  1..10000;
     N:NB;
      PROCEDURE  CRIVELLO (N: IN  NB);
          TYPE  CR  IS  ARRAY (2..N)  OF  BOOLEAN := (2..N =>TRUE);

         CRIV :CR;
         RESTO:INTEGER:=N-1;
         K:2..N;
         L:INTEGER;
          BEGIN  -- CRIVELLO

WHILE  RESTO/=0  LOOP
  K:=2
     WHILE   NOT  CRIV(K)  LOOP  -- ricerca del piu' piccolo
                                    ancora presente

              K:=K+1; END  LOOP ;
        PUT(K); -- K e' primo
        L:=K; -- si elimineranno i suoi multipli
         WHILE  L <=N  LOOP
              IF  CRIV(L)  THEN
                   CRIV(L):=FALSE;
                   RESTO:=RESTO-1;
              END  IF ;
             L:=L+K; -- multiplo successivo
         END  LOOP ;
       END  LOOP ;
    END  CRIVELLO;
BEGIN  -- programma principale
    GET (N);
    CRIVELLO(N);
END  ERA;
```

Vediamo che il parametro inserito (IN) N dalla procedura è servito come dimensione variabile per il crivello.

Si noti anche la disposizione. Questo programma è completo tranne che manca nell'intestazione di ERA una clausola USE che dichiara il package in cui trovare le procedure d'entrata-uscita GET e PUT.

ESERCIZIO 3.17: Che differenza ci sarebbe se fosse scritto:

```
TYPE CR IS ARRAY(2...N) OF BOOLEAN;
CRIB:CR:—(2..N =>TRUE);
```

I TIPI ACCESS

I tipi ACCESS consentono di gestire dinamicamente la memoria mediante puntatori, in modo analogo al PASCAL.

Un tipo ACCESS si dichiara con:

```
TYPE TA IS ACCESS specificazione del tipo
```

Ove si dichiara che gli oggetti di tipo TA saranno dei puntatori capaci di accedere a degli oggetti del tipo specificato.

Esempio:

```
TYPE TA IS ACCESS ARRAY(1..3) OF FLOAT;
P,Q:TA;
```

P e Q sono ora pronti ad accedere a dei vettori di tre reali. Non vi è pertanto nessun oggetto creato. Un oggetto sarà creato (cioè l'area di memoria necessaria sarà allocata) quando si sarà eseguito **l'allocatore:**

```
P:= NEW TA((0.5,2.0,8.24));
```

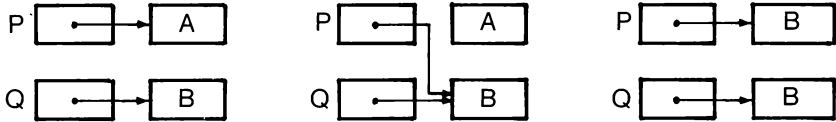
che riserva l'area di memoria necessaria per un vettore di tre reali. L'indirizzo di questo vettore è messo nella variabile P. Anche lo stesso vettore si designa con P.ALL (sostituisce la notazione di Pascal P↑).

P.ALL ha per valore (0.5,2.0,8.24)

Si può designare una componente, per esempio:

P.ALL(3) (che vale 8.24). Ma si può anche scrivere P(3).

Non confondere P.ALL:=...che dà un valore all'oggetto e Q:=P che dà un valore al puntatore Q. Dopo questa istruzione, Q punta verso lo stesso oggetto di P.

Esempio:

I puntatori possono prendere il valore NULL che significa *"punto verso nessuna parte"*.

La notazione ALL si spiega per il fatto che i puntatori sono sovente utilizzati con dei RECORD. Con il tipo usato nel paragrafo precedente, si potrebbe avere:

TYPE PT IS ACCESS IMPIEGATO;
P:PT;
P:= NEW IMPIEGATO (("C","MARTINELLI"...));

A questo punto, P.NOME è il nome dell'impiegato, P.SITFAM è la sua situazione familiare. Naturalmente P.ALL designa tutti gli impiegati.

NEW può specificare un vincolo sul tipo dell'oggetto puntato senza arrivare a dare completamente il valore dell'oggetto.

L'attributo PT'STORAGE-SIZE è la dimensione della memoria (in parole) riservata per l'allocazione di tutti gli oggetti di tipo PT.

DEFINIZIONI RECIPROCAMENTE RICORSIVE

L'oggetto può essere un RECORD di cui alcuni elementi possono essere dei puntatori verso questo tipo d'oggetto. Questa situazione si presenta in modo naturale quando si vogliono trattare delle liste o delle ramificazioni.

In una lista, l'elemento è formato dall'informazione propriamente detta e da un puntatore verso l'elemento successivo. In un albero, l'elemento è formato dall'informazione propriamente detta e da puntatori verso i vicini. Il puntatore dell'ultimo elemento della lista varrà evidentemente NULL. Ci

sarà, all'esterno un puntatore supplementare verso il primo elemento della lista.

Questo dà alla definizione un carattere ricorsivo. La definizione dell'elemento fa appello al puntatore, che è definito in funzione dell'elemento. ADA risolve il problema permettendo una definizione momentaneamente incompleta:

```
TYPE  ELEMENTO; -- momentaneamente incompleto
TYPE  PTR IS  ACCESS ELEMENTO;
TYPE  ELEMENTO IS  -- ora, la definizione completa
      RECORD
        INFO:tipo voluto;
        SEGUENTE:PTR;
      END  RECORD ;
     PARTENZA:PTR;
```

Si crea il primo elemento della lista con:

PARTENZA:= NEW ELEMENTO (valore info, NULL);

ESERCIZIO 3.18: Si è formata una lista come qui sopra. Stampate gli elementi nell'ordine della lista.

ESERCIZIO 3.19: L'informazione della lista è del tipo STRING. La lista è ordinata secondo l'ordine alfabetico, cioè P.INFO<P.SEGUENTE.INFO. Inserite nel posto giusto l'elemento il cui .INFO sarà la stringa K.

Si possono costituire delle matrici di puntatori:

```
TYPE  PTR  IS   ACCESS  tipo oggetto;
TYPE  TABP  IS   ARRAY (1..N) OF  PTR;
P:TABP;
```

P(I). ALL è l'elemento puntato dal puntatore Yesimo.

Ciò è molto utile per fare una classificazione. Gli scambi messi in gioco dalla classificazione saranno effettuati sui puntatori, cosa che fa spostare meno informazioni essendo gli oggetti puntati ingombranti.

LIBERAZIONE DELL'AREA DI MEMORIA

In Pascal, l'area di memoria occupata da una variabile puntata è esplicitamente liberata chiamando la procedura standard DISPOSE.

In ADA si preferisce la liberazione automatica che ha luogo da quando l'oggetto diventa inaccessibile, per esempio all'uscita del blocco, o della procedura che lo definisce.

Secondo il compilatore (e a seconda che si disponga di un'area di memoria molto o poco estesa), effettivamente questa liberazione ha luogo oppure no.

Altrimenti, si può con il PRAGMA:

PRAGMA CONTROLLED(nome del tipo cui si accede);

chiedere che la liberazione abbia luogo soltanto quando si esce dalla portata della definizione del tipo ACCESS.

Infine, si può fare una liberazione esplicita con l'istanziazione della procedura generica UNCHECKED-DEALLOCATION:

```
PROCEDURE FREE IS NEW
UNCHECKED-DEALLOCATION(ELEMENTO,PTR);
                          ↑        ↑
                tipo oggetto  tipo ACCESS
```

Allora FREE(P) fa P:= NULL e libera il posto che era occupato da P. ALL.

Eccoci arrivati al termine del capitolo sui tipi di dati. Si può dire che si tratta di uno dei punti forti in ADA, dove i creatori hanno sfruttato fino in fondo le potenzialità che i tipi potevano offrire.

D'altra parte, si ha il vantaggio di una migliore coerenza e di una migliore portabilità che non in Pascal: maggiori possibiltà di trattamento dei reali che fan sì che ADA sia un buon candidato per soppiantare il FORTRAN nei calcoli scientifici, effettive possibilità di trattare le stringhe di caratteri, ecc...

Uno dei concetti fondamentali della definizione dei dati in ADA è la parametrizzazione. Si possono definire dei tipi non vincolati che si vincolano in seguito a seconda della necessità. Un altro esempio molto efficace è la presentazione dei discriminanti dei RECORD come parametri.

Gli altri interventi che ora vedremo riguardano il fatto che i parametri effettivi passati ad una procedura possono vincolare il tipo di questi parametri (è l'esatta generalizzazione delle dimensioni variabili nei sottoprogrammi in FORTRAN), e che i *tipi* possono essere dei parametri negli oggetti generici, specialmente nei package.

CAPITOLO 4

SOTTOPROGRAMMI E PACKAGE MODULARITÀ-COMPILAZIONE SEPARATA-GENERALITÀ

PROCEDURE E FUNZIONI

Le procedure e le funzioni sono lo strumento essenziale della modularità dei linguaggi classici. Esse esistono in ADA, ma ADA ha altri accorgimenti per ottenere la modularità, in particolare i package.

Come tutti i linguaggi, ADA ha le due categorie di sottoprogrammi: le procedure che sono chiamate da un'istruzione di chiamata (che si riduce in ADA alla semplice citazione del nome della procedura e dei parametri), e le funzioni che restituiscono un risultato e sono chiamate dall'interno di un'espressione aritmetica.

Una procedura è specificata da:

```
PROCEDURE  nome(parametri)  IS
    dichiarazioni locali alla procedura
       BEGIN
         .
         .
         .
      istruzioni della procedura
         .
         .
         .
       END  nome;
```

Una funzione è specificata da:

```
FUNCTION  nome(parametri) RETURN  specificazione di sotto-tipo  IS
dichiarazioni
     BEGIN
       .
       .
       .
     END  nome;
```

Il flusso delle istruzioni della funzione deve terminare con un'istruzione RETURN espressione dove l'espressione è del sottotipo specificato nella dichiarazione; è il valore di questa espressione che è sostituito nell'espressione aritmetica di chiamata.

È facoltativo aggiungere il nome del sottoprogramma dopo END, ma ciò migliora la leggibilità.

GLI ARGOMENTI

La lista di argomenti o parametri ha la forma:

(nome1,nome2,nome3:modo tipo;nome4... :modo tipo;...)

Si possono raggruppare i parametri che hanno lo stesso modo e lo stesso tipo.

Il modo può essere:

IN (oppure omesso) : argomento fornito al sottoprogramma. Si comporta come una costante nel sottoprogramma, quindi non può essere modificato.

OUT : argomento restituito come risultato dal sottoprogramma.

IN OUT : argomento misto: fornito al sottoprogramma poi restituito dopo la modificazione.

Esempio:

```
FUNCTION QUADRATO (X: IN FLOAT) RETURN FLOAT IS
     BEGIN
        RETURN X**2 ;
END QUADRATO ;
```

può essere chiamata da:

```
Y,Z:FLOAT;
  A:=QUADRATO(Z)+QUADRATO(5.0*Y+4.0);
```

```
PROCEDURE DOPPIO (K: IN OUT INTEGER) IS
   BEGIN
     K:=2*K;
   END DOPPIO;
```

può essere chiamato da:

```
DOPPIO(I);
```

Come si sa, la gestione degli argomenti può essere fatta con copiatura degli argomenti forniti alla chiamata (per un parametro OUT la copia ha luogo all'atto del ritorno, per IN OUT c'è una copia alla chiamata e la copia inversa al ritorno), oppure con la trasmissione degli indirizzi che rendono le variabili dell'ambiente chiamante accessibili al sottoprogramma.

ADA non specifica il metodo che deve essere usato, cosa che consente al compilatore di scegliere il metodo che vuole e di attenervisi o anche di scegliere un metodo o l'altro a seconda del caso per fare certe ottimizzazioni. Di conseguenza, ogni scrittura che fa delle ipotesi sul metodo seguito non è corretta in ADA, mentre può essere corretta in un altro linguaggio che permetta al programmatore di specificare per ogni argomento quale metodo vuole.

Si può pensare che avrebbe potuto essere previsto un PRAGMA specifico, dicendo al compilatore *"per l'argomento X, procedere per copia; per Y, procedere per riferimento".*

I parametri IN possono essere accompagnati da un valore per default:

Esempio:

```
FUNZIONE QUADRO(X: IN FLOAT:=0.0)...
```

A questo punto, l'argomento non ha bisogno di essere fornito al momento della chiamata. Prenderà il valore per default. Se si vuole che prenda un valore diverso dal valore per default, naturalmente bisogna fornirlo.

Quando un sottoprogramma è senza parametro, (può succedere, ad esempio nel caso di una procedura il cui solo ruolo è di agire sulle variabili globali, o una funzione che — come RANDOM — non fa che restituire il suo risultato), oppure quando non vi sono da fornire dei parameri perché sono tutti presi per default, la chiamata deve essere presentata sotto la forma nome( ): il nome deve essere seguito da una coppia di parentesi vuote.

Esempio:

```
A:=B*RANDOM( )+C;
```

Altrimenti, gli argomenti presentati al momento della chiamata devono concordare in numero (tranne se alcuni sono per default), ordine (tranne se sono forniti per nome) e tipo con gli argomenti indicati nella specificazione del sottoprogramma.

Non c'è bisogno di concordanza dei nomi, e ciò è fattore di modularità. Potete utilizzare un sottoprogramma scritto da qualcun altro; tutto ciò che avete bisogno di sapere, è che questo sottoprogramma vi fornisce un risultato di questo o quel tipo. Poco vi interessa del nome usato da chi ha scritto il sottoprogramma, scegliete il nome che volete.

Viceversa, la concordanza di tipo è imperativa. E questa concordanza non può essere veramente assicurata in ADA che mediante i nomi, ecco perché i tipi anonimi sono pericolosi:

```
A: ARRAY (1..10) OF  FLOAT;
PROCEDURE  PROC(X: ARRAY  1..10  OF  FLOAT) IS
PROC(A);  --  chiamata  illegale  : A e X non hanno lo stesso
tipo
```

mentre:

```
TYPE  VETT  IS  ARRAY  1..10  OF  FLOAT ;
A:VETT;
PROCEDURE  PROC(X:VETT) IS
      .
      .
      .
PROC(A);
```

è corretto

ESERCIZIO 4.1: Nell'esempio precedente, il modo dell'argomento X non è precisato. È corretto?

Gli argomenti nel sottoprogramma figurano sempre sotto forma di variabili. Nella chiamata, gli argomenti OUT e IN OUT compaiono sotto forma di variabili. Gli argomenti IN possono essere forniti sotto forma di variabili, costanti o ogni altra espressione aritmetica: quest'ultima sarà valutata e il suo valore sarà attribuito all'argomento:

Esempio:

```
TYPE  V  IS  ARRAY (1..3) OF  FLOAT;
PROCEDURE  PROC(X: IN  V;Y: IN  FLOAT) IS ...
W:V;
T:FLOAT;
PROC(W,3*T+4.0);
PROC(W,5.0);
PROC((1.0,2.0,3.0),T);
```

sono dei richiami legali.

Si può richiamare il nome degli argomenti quando li si fornisce:

PROC(X=> (1.0,2.0,3.0),Y=>5);

Ciò è in leggera contraddizione con la regola dell'omissione del nome e, quindi, sarà poco utilizzato per dei sottoprogrammi di biblioteca. In compenso, migliora molto la leggibilità quando i nomi sono conosciuti.

Si possono mescolare i parametri dati in funzione dell'ordine nella lista, con i parametri dati per nome (e per i quali l'ordine non conta):

PROC(Y=> 5,X=> (...));

Se si mescolano, i parametri dati senza nome devono essere i primi, e seguire l'ordine; a partire dal momento in cui si danno le associazioni per nome,anche tutti i parametri lo devono essere:

```
PROCEDURE  P(A,B,C,D: IN  tipo) IS ...
P(AA,BB,C=> CC,D=> DD); e' corretto
P(AA,BB,C=> CC,DD); non e' corretto
```

Certi parametri IN possono avere un valore per default, quando i parametri non sono forniti.

```
PROCEDURE  PROC(X: IN  V;Y: IN  FLOAT:=0.0) IS ...
       PROC(X=> W); da' alla Y il valore 0.0
    PROC(X=> W,Y=>5.0); da' a Y il valore di 5.
```

In questo caso, è molto più chiaro fornire i parametri col nome.

Abbiamo visto nell'esempio del crivello di Eratostene nel capitolo precedente (p.53) che la procedura chiamata è presentata per mezzo delle dichiarazioni della procedura chiamante. Gli argomenti e gli identificatori dichiarati nella procedura chiamata sono locali a questa procedura, gli altri sono globali, (rivedere p. 21 il problema della portata degli identificatori). In ADA questa presentazione non è la sola a permettere la compilazione separata (cf. p. 89).

ESERCIZIO 4.2: Si è formata una lista come negli esercizi 3.18 e 3.19 ove l'informazione è una STRING (1...10). Scrivere una FUNCTION booleana che sia TRUE se la stringa K è presente nella lista. Scrivere una funzione analoga ma di tipo puntatore. Il risultato è il puntatore verso l'elemento della

lista che contiene K, o NULL se K non è nella lista. Fare degli esempi di chiamata.

ESERCIZIO 4.3: Completare l'esercizio 3.19 per costituire una procedura. Fare degli esempi di chiamata.

TRASMISSIONE DI MATRICI DI DIMENSIONI VARIABILI

Un argomento IN può essere di un tipo non vincolato. Il tipo diventerà vincolato al momento della chiamata quando sarà fornito un valore. Il caso in cui ciò avviene più frequentemente è quando l'argomento è una matrice dai limiti indeterminati.

Ecco un prodotto scalare di due vettori:

```
PROCEDURE  PROD  IS
      TYPE  VETT  IS   ARRAY (INTEGER  RANGE  <>) OF  FLOAT;
     PR:FLOAT;
      PROCEDURE  PRODSCAL(X,Y: IN  VETT;XY: OUT  FLOAT) IS
      BEGIN
        XY:=0.0;
         FOR  I  IN  X'RANGE  LOOP
        XY:=XY+X(I)*Y(I);
         END   LOOP ;
END  PRODSCAL;
BEGIN
  PRODSCAL((1.0,2.0,3.0  ,(4.0,5.0,6.0),PR);
  PUT(PR);
   .
   .
   .
   .
END  PROD;
```

La chiamata a PRODSCAL con i vettori 1,2,3, e 4,5,6 fissa la dimensione a 3. Anche tutte le altre chiamate dovranno essere fatte con questa dimensione. Le dimensioni non sono variabili per impedire che si possa chiamare la procedura con dei vettori di diverse dimensioni. Ma così come sono consentono di scrivere il sottoprogramma e di incorporarlo una volta per tutte in una biblioteca, a compilazione separata.

La procedura così come è scritta ha una lacuna; bisognerebbe assicurarsi che X'FIRST=Y'FIRST e X'LENGTH=Y'LENGTH altrimenti il prodotto non è possibile. In ADA preliminare, si disponeva di una istruzione particolare:

```
ASSERT X'LENGTH=;Y'LENGTH;
```

che faceva scattare un'eccezione se la condizione non era soddisfatta.

Infatti, è sufficiente scrivere (subito dopo il BEGIN):

```
IF X'LENGTH/=Y'LENGTH THEN RAISE ERRORE-LUNGHEZZA;
END IF;
```

e, naturalmente, fornire il trattamento dell'eccezione ERRORE-LUNGHEZZA; (vedere il capitolo sulle eccezioni).

ESERCIZIO 4.4: Scrivere una procedura capace di scambiare due matrici M1 ed M2.

ESERCIZIO 4.5: Scrivere una procedura che calcoli il prodotto C di due matrici A e B.

RICORSIVITÀ

Come in ogni linguaggio sufficientemente evoluto, una funzione o una procedura possono chiamare se stesse.

Ecco una funzione fattoriale definita da fatt(n)≡n*fatt(n—1):

```
FUNCTION  FACT(N:INTEGER) RETURN  INTEGER  IS
    BEGIN
       IF  N>1  THEN   RETURN  N*FACT(N-1);
                 ELSE   RETURN  1;
       END   IF ;
    END  FACT;
```

ESERCIZIO 4.6: Torri di Hanoi

Si dispone di tre picchetti. Sul picchetto di sinistra (1), sono accatastati N dischi (forati) di diametro decrescente. Si tratta di trasportarli sul picchetto 2 obbedendo ad alcune regole:

1) *si sposta un disco alla volta;*
2) *un disco è sempre infilato su di un picchetto;*
3) *su di un picchetto, un disco può essere sovrapposto ad un altro soltanto se è di diametro più piccolo.*

Il problema è essenzialmente ricorsivo. Infatti:

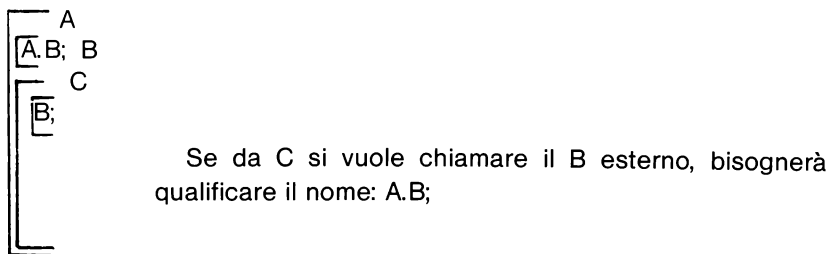
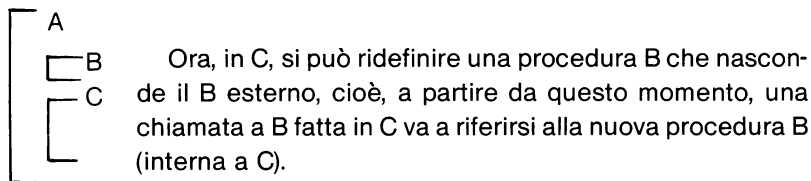
1) *Sappiamo spostare 1 disco. Basta scrivere:*
   numero picchetto partenza => numero picchetto arrivo.
   Si avrà così la lista dei movimenti da effettuare.
2) *spostare n dischi da i a j, significa:*
   spostare n-1 dischi da i al terzo picchetto
   spostare 1 disco da i a j
   spostare gli n-1 dischi dal terzo picchetto a j

Scrivere il programma corrispondente.

SOVRACCARICO

Sia data una procedura o un blocco A nel quale sono definite le procedure B e C. Dopo C, si può chiamare B.

Ora, in C, si può ridefinire una procedura B che nasconde il B esterno, cioè, a partire da questo momento, una chiamata a B fatta in C va a riferirsi alla nuova procedura B (interna a C).

Se da C si vuole chiamare il B esterno, bisognerà qualificare il nome: A.B;

Questa è la regola classica di portata delle definizioni. Tuttavia, in ADA, la qualificazione offre una flessibilità supplementare.

Anche nel blocco A, si può definire un secondo B, diverso dal primo per il tipo degli argomenti (e/o il loro numero).

```
PROCEDURE B(X:T1) IS...
PROCEDURE B(Y:T2) IS...
```

Si dice che il secondo B è un **sovraccarico** del primo.

ADA distinguerà le chiamate basandosi sul tipo:

```
M:T1;
N:T2;
B(M); -- chiamata del primo B
B(N); -- chiamata del secondo
```

Ogni ambiguità va eliminata; le ambiguità si producono più frequentemente se una stessa costante è sovraccaricata su più tipi:

```
TYPE  T1  IS  (IJK);
TYPE  T2  IS  (K,L,M);
B(I);  -- nessuna ambiguita': e' il primo B
B(L);  -- nessuna ambiguita' : e' il secondo B
B(K);  -- non corretto
```

Si evita l'ambiguità qualificando l'argomento: B(T1'(K)); —— è il primo B.

Sovraccarico degli operatori:

Esponiamo ora una possibilità straordinaria offerta da ADA. Il nome di una funzione può essere identico ad un operatore standard del linguaggio. La definizione della funzione avrà come effetto di sovraccaricare questo operatore quindi di definirlo per tipi diversi da quelli per i quali è definito normalmente.

Per un operatore monadico, occorre un argomento che svolgerà il ruolo dell'operando di destra:

```
FUNCTION "—" (A:COMPLEX) RETURN COMPLEX IS
```

definirà come ottenere l'opposto di un numero complesso.

Per un operatore diadico, occorrono due argomenti: il primo svolgerà il ruolo di operando di sinistra, il secondo di operando di destra:

```
FUNCTION "*" (A,B:MATRICE) RETURN MATRICE IS
```

definirà il prodotto di matrici.

La novità consiste nel fatto che la nuova accezione dell'operatore viene

usata nelle espressioni aritmetiche nel modo consueto:

C:=A*B;

effettua un prodotto di matrici se C,A e B sono delle matrici. Bene inteso valgono sempre le regole di concordanza dei tipi e di non ambiguità.

Terminiamo la definizione del prodotto di matrici. È sufficiente trasformare l'esercizio 4.5 in funzione e cambiare il nome:

```
FUNCTION  "*"(A,B: IN  MATRICE)  RETURN  MATRICE  IS
      Z:FLOAT; C:MATRICE;
       BEGIN
          IF  A'RANGE(2)/=B'RANGE  THEN  RAISE  NUMERIC ERROR;
          END  IF ;-- si genera un'eccezione predefinita
          FOR  I  IN  A'RANGE  LOOP
              FOR  J  IN  B'RANGE  LOOP
                Z:=0.0;
                 FOR  K  IN  B'RANGE  LOOP
                    Z:=Z+A(I,K)*B)K,J);
                 END  LOOP ;
                C(I,J):=Z;
              END  LOOP ;
          END  LOOP ;
          RETURN  C;
END  "*";
```

SOTTOPROGRAMMI ARGOMENTO

Un sottoprogramma può essere argomento di una procedura o di una funzione?

Ciò è molto utile per la situazione seguente. Possiamo avere scritta una funzione di integrazione o una procedura di tracciamento della curva rappresentativa di una funzione. Uno degli argomenti della funzione d'integrazione, per esempio, dirà di quale funzione si vuole l'integrale. Ciò è possibile in Fortran e in alcune versioni del PASCAL, mentre è impossibile in ADA dove né un sottoprogramma né un tipo possono essere argomento di sottoprogrammi.

Ma — ed è assolutamente necessario — occorre pure una soluzione al nostro problema d'integrazione. ADA la fornisce grazie agli elementi generici che consentono di fare dipendere un sottoprogramma o un package da certi sottoprogrammi o anche da certi tipi (cf. p. 93).

IL PRAGMA INLINE

Normalmente, ed è anche uno dei vantaggi decisivi dei sottoprogrammi, un sottoprogramma è scritto in memoria una sola volta e, ad ogni chiamata, si ha un salto al suo indirizzo. Ciò fa guadagnare spazio di memoria, ma fa perdere un po' di tempo durante l'esecuzione.

Se si specifica (nella stessa parte dichiarativa della definizione del sottoprogramma):

```
PRAGMA INLINE(NOME);
```

il sottoprogramma nome sarà caricato in memoria all'indirizzo di ogni chiamata, in sostanza lo stesso meccanismo dei macro nell'assembler.

Ciò fa risparmiare tempo ma consuma della memoria. L'effetto prodotto all'esecuzione, in compenso, non è modificato.

I PACKAGE

Ecco una delle maggiori innovazioni di ADA. Un package è un insieme di risorse: tipi, variabili, sottoprogrammi che concorrono ad un tipo di trattamento che è messo a disposizione del programma.

Il più delle volte, si utilizzerà un package già pronto di cui si conosce l'esistenza. In conclusione, i package di biblioteca generalizzano la nozione di funzione di biblioteca e l'arricchiscono: con un package non si forniscono soltanto i mezzi d'esecuzione di un algoritmo (i sottoprogrammi), ma anche i dati associati, il loro tipo, la loro organizzazione.

Era normale che ADA offrisse questo arricchimento, poiché uno dei progressi di ADA è di porre una grande attenzione ai dati e non più soltanto ai trattamenti.

Ma la grande innovazione del meccanismo dei package è di separare la specificazione delle proprietà del package dalla realizzazione propriamente detta di queste proprietà. E difatti un package comprende il più delle volte due parti: la specificazione e il **corpo** (body) che possono essere compilati separatamente.

Ciò è molto importante nello sviluppo di un programma in analisi discendente: sapendo che ho questa o quella necessità, li specifico nella parte specificazione del mio package e scrivo, o mi faccio scrivere, il corpo del package.

Un altro aspetto interessante è che, se, per ritornare all'esempio del Capitolo I, compro un package di gestione di numeri complessi, tutto ciò che mi interessa, è di sapere di quali operazioni dispongo sui miei numeri complessi. Il modo in cui queste operazioni sono realizzate mi interessa poco. Dunque, in un tal caso, soltanto la parte specificazione sarà comunicata all'utilizzatore; **il corpo** non lo sarà più.

Ma si può andare ancora più lontano. I dettagli di rappresentazione dei numeri complessi scelti dalla software house non mi interessano. Ecco perché, come lo si è visto nel Capitolo 1, il tipo dei numeri complessi sarà reso PRIVATE ed a me, che sono il cliente, non saranno comunicati i dettagli della struttura.

ADA consente di controllare totalmente le operazioni che l'utilizzatore può effettuare su di un oggetto. Questo è importante nei problemi di sicurezza e di protezione. Per esempio, in un sistema di gestione con parola d'ordine, non solamente la struttura interna delle parole d'ordine non sarà comunicata all'utilizzatore, ma addirittura gli si impedirà ogni operazione diversa da quelle poche necessarie.

Ciò si ottiene con il tipo LIMITED PRIVATE dove anche l'assegnazione e il test di uguaglianza sono vietati (se fossero autorizzati, l'utilizzatore potrebbe arrivare a decodificare la struttura).

Un package ha dunque la rappresentazione seguente:

```
PACKAGE nome IS
   dichiarazioni;
[PRIVATE                                    specificazione
   dichiarazione nascosta;]
END [nome];

PACKAGE BODY nome IS
   dichiarazioni;
[BEGIN                                      corpi del
   istruzioni di inizializzazione;]         package
END [nome];
```

Molto raramente il PCKAGE sarà privato del corpo. A questo punto la sua specificazione si ridurrà a delle dichiarazioni di tipo e di variabile. In

questo caso, il package non serve che da raggruppamento di dichiarazioni e l'attivazione del package mediante:

USE nome del package;

non sarebbe molto differente dall'inclusione del testo delle dichiarazioni.

Una tale inclusione potrebbe essere fatta con il PRAGMA:

PRAGMA INCLUDE (nome del file contenente il testo);

Questo PRAGMA può essere interessante per includere là dove si vuole dei brani di testo sorgente scritti una volta per tutte. Non vi ritorneremo, e così pure sui package ridotti a delle dichiarazioni.

La parte istruzioni del corpo del package è, anch'essa, raramente utilizzata e non contiene che delle istruzioni di inizializzazione dei dati che possono essere effettuate anche in modo differente.

Nella parte specificazione, le funzioni e le procedure sono introdotte solo da una loro intestazione che precisa il tipo degli argomenti e dei risultati. È nella parte dichiarazioni del corpo del package che figura il corpo dei sottoprogrammi con l'intestazione ripetuta e le istruzioni propriamente dette.

I sottoprogrammi sono molto spesso dei sovraccarichi degli operatori standard, estendendo questi operatori a degli operandi appartenenti ai nuovi tipi introdotti nel package.

La parte dichiarazioni del corpo del package può contenere inoltre delle dichiarazioni di variabili locali al package. Queste variabili hanno una proprietà interessante: formano degli **oggetti persistenti**, cioè conservano il loro valore fra due chiamate ad una procedura del package.

Queste variabili non sono accessibili dall'esterno del package. Esse sono accessibili solo a partire dalle procedure del package, cosa che dimostra chiaramente che il corpo del package è "nascosto".

Le istruzioni del corpo di un package possono terminare con definizioni di trattamento di eccezioni (cfr. Capitolo 7).

Infine in ciascuno degli END della specificazione e del corpo del package il richiamo del nome del package è facoltativo. Si può non metterlo, o metterlo a entrambi, oppure metterlo ad uno e non all'altro, ma facilita comunque la lettura.

Rinviamo il lettore all'esempio molto chiaro dei numeri complessi nel Capitolo I. Quando c'è un tipo privato, questo tipo deve essere dichiarato nella parte visibile:

```
PACKAGE ....  IS
    .
    .
    .
    TYPE  P  IS   PRIVATE ;
    .
    .
    .
PRIVATE
    .
    .
    .
    TYPE  P  IS  .... ;
```

e totalmente definito nella parte privata della specificazione.

Quando si ha una costante di tipo privato, deve essere annunciata sotto forma di costante differita, e definita totalmente (inizializzata) nella parte privata:

```
PACKAGE  ...  IS
    TYPE  P  IS   PRIVATE ;
   C: CONSTANT  P;
PRIVATE
    TYPE  P  IS  ...;
   C: CONSTANT  P :=.... ;
```

Ecco un altro esempio. Formiamo un package di gestione di risorse (per esempio dei file). L'utilizzatore dispone di tre operazioni possibili (che) daranno luogo a tre procedure):

NUOVOUT: immatricolazione di un nuovo utilizzatore. Questi fornisce la parola d'ordine che sarà messa in repertorio (e che egli dovrà fornire ad ogni utilizzazione del sistema), e riceve un numero di utente.

GETRISORSA: l'utente si fa assegnare una risorsa. L'utente dà il suo numero di utilizzatore, la sua parola d'ordine e riceve un nome di risorsa.

ACCESSORISORSA: per utilizzare una risorsa che si è fatta attribuire, l'utilizzatore indica il suo numero, la parola d'ordine e il nome della risorsa. Sicuramente, ogni disaccordo fra il numero e la parola d'ordine sarà punito.

Il tipo RISORSA sarà LIMITED PRIVATE in modo che le manipolazioni di risorse siano solo quelle previste da GETRESSOURCE e ACCESRESSOURCE.

La parte visibile della specificazione del package sarà allora:

```
PACKAGE  GESTIONE-RISORSE  IS
     TYPE  PASSWORD  IS  STRING(1..10); -- tipo visibile
     TYPE  RISORSA  IS  LIMITED  PRIVATE ;
     PROCEDURE  NUOVOUT(PAROLA-PASSA: IN  PASSWORD);
     PROCEDURE  GETRISORSA(NUT: IN  INTEGER;PAROLA-PASSA: IN
PASSWORD;
                  NOMR: OUT  RISORSA) ; PROCEDURE
ACCESSORISORSA(NUT: IN  INTEGER; PAROLA-PASSA: IN  PASSWORD;NOMR:
IN RISORSA);
```

—— può darsi che vi siano altri argomenti che
—— dipendono dall'utilizzazione della risorsa.

Solamente questa è fornita e documentata all'utilizzatore.

La parte privata della specificazione è:

```
PRIVATE
     TYPE  RISORSA  IS             -- descrizione della
          RECORD                   -- rappresentazione
             NOME:STRING(1..10);   -- interna della risorsa.
             NUMERO:INTEGER;       -- Dipende
          END  RECORD ;            -- dal tipo di uso
END  GESTIONE-RISORSE;            -- ma, comunque, non e' noto
                                   -- all'utente.
```

Il corpo del package descriverà i dati utilizzati all'interno del package e quindi sconosciuti all'esterno. In particolare ci sarà una tabella degli utilizzatori immatricolati, con le loro parole d'ordine e una tabella delle risorse attribuite.

```
PACKAGE  BODY  GESTIONE-RISORSE  IS
   MAX-UT: CONSTANT  INTEGER:=100; -- n. max. utenti
   MAX-RES:  CONSTANT  INTEGER:=10;  --  max.  risorse  per
utente
   NB-UT:INTEGER; -- numero d'utenti immatricolati
    TYPE  UTIL  IS
```

```
     RECORD
        PAROLA:PASSWORD; -- parola d'ordine
        NUMRI:1..MASS-R  IS ;  -- nr di risorse attribuite


     END   RECORD ;
UTILIS: ARRAY (1..MASS-UT) OF  UTIL;
RIS:  ARRAY (1..MASS-UT,1..MASS-RIS) OF  RISORSA;
```

Ciò delinea già la gestione che ne sarà fatta. Bisogna però che sia inaccessibile all'utilizzatore, altrimenti potrebbe falsificarla: aver più risorse del massimo concesso, accedere alle parole d'ordine degli altri...

Ecco l'abbozzo delle procedure:

```
PROCEDURE  NUOVOUT(PAROLA D'ORDINE: IN  PASSWORD) IS
   BEGIN
    NR-UT:=NR-UT+1;
     IF  NR-UT>MASS-UT  THEN   RAISE  eccezione voluta;
     END   IF ;
    PUT("IL VOSTRO NUMERO SARA'"); PUT(NR-UT);
    UTILIS(NR-UT).PAROLA:=PAROLA D'ORDINE; -- registra la parola
                                                d'ordine
    UTILIS(NR-UT).NUMRI:=0; -- non ci sono ancora di risorse
   END  NUOVOUT;


PROCEDURE  GETRISORSA(NUT: IN  INTEGER; PAROLA-D'ORDINE:
                      IN  PASSWORD; NMR:  OUT  RISORSA) IS

     NR:1..MASS-RIS;
      BEGIN
         IF  NUT  NOT   IN  UTILIS'RANGE  THEN  protesta in
                                              modo conveniente;
         END   IF ; -- non e' il numero buono
         IF  UTILIS(NUT).PAROLA/-PAROLA D'ORDINE  THEN  protesta;
         END   IF ; -- passa la parola d'ordine valida
     NR:=UTILIS(NUT).NRI+1; -- ci sara' automaticamente eccezione
                                 se mass-ris e' superata
     NUMR:= fabbricazione adeguata della risorsa;
     UTILIS(NUT).NUMR:=NR;
     RIS(NUT,NR):=NUMR; -- registra la risorsa
   END  GET RISORSA;
```

```
PROCEDURE  CEESO RISORSA(NUT: IN  INTEGER;PAROLA-D'ORDINE:
                  IN  PASSWORD; NUMR: IN  RISORSA;...) IS
    BEGIN
   -- stessi test su NUT e PAROLA-D'ORDINE
   -- test che NUMR figura fra le risorse registrate
    FOR  I  IN  1..UTILIS(NUT).NUMR  LOOP
        IF  RIS (NUT,1)=NUMR  THEN ...
   -- si ha il diritto all'assegnazione e al test d'uguaglianza
   -- all'interno del package
   -- accesso alla risorsa
    END  ACCESSORISORSA;
END  GESTIONE-RISORSE;
```

Una utilizzazione potrà presentarsi nel modo seguente :

```
PROCEDURE  UTILIZZAZIONE  IS
             USE  GESTIONE- RISORSE;
            MIERISORSE: ARRAY (1..5)  OF  RISORSA; -- la matrice
                                                    delle mie risorse

           BEGIN
            NUOVOUT ("TAGADAGADA");
            ( IL VOSTRO NUMERO SARA' 5    )

poi :

            GETRISORSA(5,"TAGADAGADA", MIE-RISORSE(1));
            -- acquisto la mia prima risorsa.
            ACCESSORISORSA(5,"TAGADAGADA",MIE-RISORSE(1));
            -- la utilizzo.
```

ESERCIZIO 4.9: Si sarebbe potuto scrivere?
R:RISORSA;
GETRISORSA(5,"TAGADAGADA",R);
MIERISORSE(1):=R;

ESERCIZIO 4.10: Si vuole costituire un package di gestione di catasta. Si ricorda che una catasta è gestita secondo la procedura LIFO (last in, first out: ultimo entrato, primo uscito). Gli elementi da accatastare saranno di un tipo T (per fissare le idee, delle stringhe di 10 caratteri). Le sole cose di cui l'utilizzatore disporrà saranno le procedure ACCATASTARE (aggiungere

un elemento al di sopra della catasta) e TOGLIERE (ritirare l'elemento dal di sopra), e le variabili booleane PILA VUOTA e PILA PIENA. TOGLIERE può essere una funzione che restituisce il valore dell'elemento tolto. Il modo in cui la catasta è gestita dal package deve essere sconosciuto all'utilizzatore. Dare la specificazione del package e il corpo nelle due ipotesi seguenti:

— *la catasta è gestita come una tabella; si conserva l'indice dell'ultimo elemento aggiunto,*

— *la catasta è gestita da allocazioni e puntatori.*

UTILIZZAZIONE DEI PACKAGE - CLAUSOLA USE

Un package è accessibile, cioè gli elementi definiti nella sua parte visibile sono accessibili, in ogni blocco inscritto nel blocco in cui il package è dichiarato; è la regola di visibilità abituale.

```
PROCEDURE  A  IS
    PACKAGE  P  IS
    X:
    END  P;
    PROCEDURE  B  IS
     3
    BEGIN --B
     1
    END  B;
   . 4
   .
   .
    BEGIN --A
     2
    END  A;
```

Nella figura qui a fianco, si può accedere all'elemento X del package P in 1 o in 2.

L'accesso si farà con un nome qualificato: P.X.

Si evita la necessità dal qualificatore P con una clausola USE P; collocata (secondo la necessità) in 3 o 4.

A questo punto, l'accesso a X si farà semplicemente sotto la forma X.

La clausola USE non ha reso X visibile, lo era già, essa non fa altro che semplificare il modo di far riferimento a X evitando di qualificarlo col nome del package.

Una clausola USE può comparire ovunque in una parte dichiarativa.

Una o più clausole USE possono rendere un nome ambiguo. Si toglie l'ambiguità ritornando alla notazione qualificata.

Un altro modo di evitare di qualificare il nome sarebbe stata una clausola RENAMES della forma:

PX :tipo RENAMES P.X; per una variabile

oppure

PROCEDURE TOTO(argomenti) RENAMES P.ZOZO; se ZOZO è una procedura del package.

Abbiamo già accennato che ogni volta che utilizziamo le procedure di ingresso-uscita standard, (GET e PUT), bisogna mettere la clausola USE del package (predefinito) che le contiene.

COMPILAZIONE SEPARATA

I sottoprogrammi, i package (e, vedremo in seguito i task) sono unità di compilazione in quanto possono essere compilate una per volta.
È questo il caso delle biblioteche d'operazioni standard, compilate separatamente dal vostro programma una volta per tutte.

ISTRUZIONE WITH

Nell'intestazione della vostra unità di compilazione, dovete precisare la lista dei package già compilati che volete utilizzare. È l'oggetto dell'istruzione WITH:

WITH nome del package [nome del package];

Per esempio, si potrà avere:

WITH ENTRATE-USCITE,ARITMETICA-REALE;
PROCEDURE MIO-PROGRAMMA IS ....

Questo unisce al contesto del programma i package che si sono supposti forniti con le biblioteche INGRESSO-USCITA e ARITMETICA-REALE.
Questa istruzione non deve essere confusa con USE.

Si mette un solo WITH nell'intestazione dell'insieme da compilare. Si pone un USE di ogni package introdotto nella parte dichiarativa di ogni

procedura che deve poter accedere agli oggetti senza nomi qualificati.

Può esservi un USE che accompagna il WITH se il programma principale (che comprende tutto il resto) della compilazione fa riferimento ad un tipo o a un (sotto)package definito in uno dei package della biblioteca.

Esempio:

```
WITH  A,B,C; USE  A;
PROCEDURE  AA  IS
   AAA:AAAA; -- il tipo AAAA e' definito in A


    USE  A-A; -- package interno definito in A
PROCEDURE  BB  IS
    USE  B,C;
    .
    .
    .
```

Soltanto dei package esplicitamente referenziati devono essere citati nel WITH.

Dato il programma 1:

```
1

PROCEDURE  PP  IS
      PACKAGE  PA  IS
      .
      .
      END  PA;
      PACKAGE   BODY  PA  IS
      .
      .
      .
      END  PA;
      PROCEDURE  SP  IS
         USE  PA;
      END  SP;
BEGIN --PP
END  PP;
```

```
2

PACKAGE  PA  IS
END  PA;

PACKAGE   BODY  PA  IS
END  PA;
WITH  PA;

PROCEDURE  PP  IS
    PROCEDURE  SP  IS
   .
   .   USE  PA;
    END  SP;
BEGIN  -- PP
END  PP;
```

Si può, se si vuole, compilare separatamente il package PA e la procedura PP. Si può anche compilare separatamente la specificazione e il corpo di PA.

Sarà sufficiente per questo fare precedere PROCEDURE PP da una clausola WITH PA; come qui sopra, sotto la parte 2.

Ogni coppia sotto la parte 2 designa un testo che può essere compilato separatamente dagli altri.

CLAUSOLA SEPARATE:

Si può fare compilare separatamente una procedura o il corpo di un package interno ad una procedura. La procedura separata o il corpo del package dovrà essere rappresentata da un'abbreviazione della forma:

PROCEDURE o FUNCTION nome(argomenti) IS SEPARATE;
PACKAGE BODY nome IS SEPARATE;

Il corpo della procedura o del package sarà fornito sotto la forma:

SEPARATE (nome del blocco da cui lo si è separato)
PROCEDURE nome IS

Esempio:

Sotto 1 è la forma non separata, sotto 2 la forma separata:

```
WITH  BIBLI,
PROCEDURE  PP  IS
      PACKAGE  PA  IS
        FUNCTION  F(X:FLOAT) RETURN  FLOAT;
        PROCEDURE  G(Y,Z:FLOAT);
      END  PA;
      PACKAGE   BODY  PA  IS
         FUNCTION  F(X:FLOAT) RETURN  FLOAT  IS
             .
             .
             .
         END  F;
         PROCEDURE  G(Y,Z:FLOAT) IS
        -- utilizza BIBLI
         END  G;
      END  PA;
      PROCEDURE  SP(A,B:FLOAT) IS
         USE  PA;
        .
        .
      END  SP;
```

```
BEGIN --PP
    .
    .
    .
END  PP;
```

2

```
PROCEDURE  PP  IS
    PACKAGE  PA  IS
      FUNCTION  F(X:FLOAT) RETURN  FLOAT;
      PROCEDURE  G(Y,Z:FLOAT);
    END  PA;
    PACKAGE  BODY  PA  IS  SEPARATE ;
    PROCEDURE  SP(A,B:FLOAT) IS  SEPARATE ;
BEGIN --PP
    .
    .
    .
END  PP;
```

```
SEPARATE (PP)
PROCEDURE  SP(A,B:FLOAT) IS
     USE  PA;
    .
    .
END  SP;
```

```
SEPARATE (PP)
PACKAGE  BODY  PA  IS
   FUNCTION  F(X:FLOAT) RETURN  FLOAT  IS
      .
      .
      .
   END  F;
   PROCEDURE  G(Y,Z:FLOAT) IS  SEPARATE ;
END  PA;
```

```
WITH  BIBLI;
SEPARATE (PP.PA)
PROCEDURE  G(Y,Z:FLOAT) IS
-- testo di G che
-- utilizza BIBLI
END  G;
```

Anche qui, i raggruppamenti marcano le quattro parti del testo che compileremo separatamente. Si dice che PA ed SP formano delle sottounità di PP e che G forma una sottounità di PA. Notate come è fornito il nome qualificato di PA nella clausola SEPARATE e che, siccome è G che si serve di BIBLI, è nella compilazione di G che il WITH corrispondente è fornito.

ESERCIZIO 4.11: Nell'esempio dell'esercizio 4.10, compilate separatamente il corpo del package e le procedure TOGLIERE e AGGIUNGERE.

ORDINE DI COMPILAZIONE

L'ordine nel quale dei moduli che dipendono gli uni dagli altri devono essere compilati risulta semplicemente dal buon senso: ogni modulo A citato nel WITH di un modulo deve essere compilato prima di B.

Ciascun corpo di sotto-programma o package deve essere compilato dopo la dichiarazione del sotto-programma o la specificazione del package. Le sottounità devono essere compilate dopo l'unità da cui sono separate.

Le stesse regole si applicano alla ricompilazione. Una modifica di una sottounità non ha bisogno di ricompilazione dell'unità. Una modificazione di una unità **può** avere bisogno della ricompilazione delle sottounità.

ELEMENTI GENERICI

Le clausole GENERIC consentono di parametrizzare un sottoprogramma, un package o un task mediante un tipo o un sottoprogramma. Anche delle variabili possono servire da parametri generici.

Un oggetto generico non è che un modello dell'oggetto reale. Per utilizzarlo, bisognerà creare un oggetto del genere, definito da dei valori determinati dei parametri. Questa creazione si chiama **istanziazione**.

Ritorniamo all'esempio del Capitolo 1 p. 12.

Abbiamo definito una procedura di scambio di due oggetti. È evidente che il procedimento è lo stesso qualunque sia il tipo degli oggetti. Il tutto è riunito per definire una procedura generica:

```
GENERIC
    TYPE  T  IS  PRIVATE ;
PROCEDURE  SCAMBIO(X,Y: IN   OUT  T) IS
   Z:T;
   BEGIN
      Z:=X;X:=Y;Y:=Z;
END  SCAMBIO;
```

Così facendo, non abbiamo definito alcuna procedura di scambio. Ora, vogliamo scambiare degli interi: **instanziamo** la procedura generica per gli interi:

```
PROCEDURE  SC-INTERI  IS   NEW  SCAMBIO(INTEGER);
SC-INTERI e' una vera procedura, utilizzabile :
SC-INTERI(I,J);
```

Si avranno tante istanziazioni della procedura generica quante se ne vogliono. Per scambiare delle matrici:

```
TYPE  MATRICE  IS   ARRAY (1..10,1..10) OF  FLOAT;
A,B:MATRICE;
PROCEDURE  SCAMAT  IS   NEW  SCAMBIO(T=>MATRICE);
  .
  .
  .
SCAMAT(A,B);
```

L'esempio dimostra che si può definire il parametro citando il suo nome con il solito segno =>. L'istanziazione di una procedura non richiama gli argomenti (si ritrovano esaminando il modello generico).

Se i nomi utilizzati da più istanziazioni sono identici, si ottengono dei sovraccarichi.

VARIABILI PARAMETRO

I parametri sono utilizzati in modo molto simile a quelli dei sottoprogrammi. Si ha una sostituzione del valore al momento dell'istanziazione. Il modo OUT è senza oggetto.

Esempio:

```
GENERIC
   MISURA:NATURALE;
PACKAGE  AZIONE-SU-MATRICI  IS
        TAB: ARRAY (1..MISURA) OF ...
```

L'istanziazione:

```
PACKAGE  MATRICI  IS  NEW  AZIONE-SU-MATRICI(100);
```

fisserà la dimensione a 100.

TIPI PARAMETRO

Un tipo parametro può essere presentato sotto le forme:

1 — TYPE nome formale IS (LIMITED) PRIVATE;
Allora potrà, all'atto dell'istanziazione essere sostituito da ogni tipo, tranne un ARRAY non vincolato.
Se LIMITED è specificato, le operazioni di attribuzione e di test di uguaglianza non hanno bisogno d'essere definite e il tipo formale potrà essere sostituito da un tipo task.

2 — TYPE nome formale IS
Potrà essere sostituito da ogni tipo discreto.

3 — TYPE nome formale IS RANGE < >;
In questo caso, potrà essere sostituito da ogni intero.

4 — TYPE nome formale IS DIGITS < >;
In questo caso, potrà essere sostituito da ogni tipo reale a virgola mobile.

5 — TYPE nome formale IS DELTA < >
Allora, potrà essere sostituito da ogni tipo reale a virgola fissa.

6 — TYPE nome formale IS ARRAY (specificazione d'indici) OF tipo;
Potrà essere sostituito da ogni tipo matrice compatibile (stesso numero d'indici, vincoli identici).

Esempi:

```
GENERIC
    TYPE  OGGETTO  IS  PRIVATE ;
    TYPE  MATRICE  IS  ARRAY (INTEGER RANGE <>) OF OGGETTO;

GENERIC
    TYPE  INDICE  IS
    TYPE  VETT  IS  ARRAY (INDICE) OF FLOAT;
```

La specificazione più generale è PRIVATE. ADA revisionato ha introdotto gli altri tipi per consentire un controllo più preciso.

ESERCIZIO 4.12: "Parametrizzare" il PACKAGE di gestione delle cataste dell'esercizio 4.10 in funzione del tipo degli elementi da accatastare e della dimensione stabilita.

SOTTOPROGRAMMI PARAMETRI

Una procedura o una funzione possono essere parametro. Questi parametri sono situati nella parte generica dopo le variabili e i tipi.

Le tre forme possibili sono:

1 — WITH PROCEDURE nome formale (argomenti);
WITH FUNCTION nome formale (argomenti) RETURN tipo;

I tipi referenziati possono essere dei parametri dello stesso generico.

Esempio:

```
GENERIC
    TYPE  REALE  IS  DIGITS <>;
    WITH  FUNCTION  F(X:REALE) RETURN  REALE;
PROCEDURE  ZOZO  IS
     .
     .
     .
     Z:=F(X)
     .
     .
     .
```

Una istanziazione sarà per esempio:

```
TYPE  R  IS  DIGITS  10;
```

```
FUNCTION  TRUCCO(X:R)  RETURN  R  IS
        .
        .
        .
END  TRUCCO;
PROCEDURE  TOTO  IS  NEW  ZOZO REALE=>R,F=> TRUCCO);
```

Sotto questa forma, l'istanziazione deve sempre fornire l'argomento procedura. Nella seconda forma, si dà una procedura per default :

2 — {WITH PROCEDURE / WITH FUNCTION} {non formale (argomenti) / non formale (argomenti)} {RETURN tipo;} {IS nome / per default}

Esempio:

```
GENERIC
WITH  PROCEDURE  A(X:REALE) IS  B;
PROCEDURE  D  IS ...
```

L'istanziazione:

```
PROCEDURE  DC  IS  NEW  D(C); fara' intervenire C.
```

L'istanziazione:

```
PROCEDURE  DB  IS  NEW  D; fara' intervenire B.
```

3 — La terza forma interviene quando il parametro è una versione di un operatore standard del sistema:

Esempio:

```
GENERIC
  TYPE  T  IS  PRIVATE ;
  WITH  FUNCTION  ''*ùù(A,B:T) RETURN  T  IS <>;
FUNCTION  QUADRATO(A:T) RETURN  T  IS
    BEGIN
      RETURN  A*A;
    END  QUADRATO;
```

Ora, si definisce:

```
   TYPE  MATRICE  IS  ARRAY (1...10,1..10) OF  FLOAT;
   FUNCTION  PRODMA(A,B: IN  MATRICE) RETURN  MATRICE  IS
  -- il prodotto di matrici
   END  PRODMA;
```

L'istanziazione:

FUNCTION QUADMAT IS NEW QUADRATO(MATRICE,PRODMA);

calcola il quadrato delle matrici, utilizzando il prodotto di matrici.

L'istanziazione:

FUNCTION INTSQR IS NEW QUADRATO(INTEGER);

calcola il quadrato degli interi. Poiché non è fornito alcun parametro per sostituire "*" e dal momento che esiste un sovraccarico standard di "*" in vigore per il tipo fornito (INTEGER), questa è la versione che si utilizza.

Siamo in grado ora di scrivere una funzione d'integrazione generale.

ESERCIZIO 4.13: Scrivere dapprima una funzione che fornisca l'integrale di A e B della funzione fissa F. Si applicherà la formula dei trapezi a N intervalli:

$$H=\frac{B-A}{N} \quad \text{integrale} \neq \frac{1}{2}\,(F(A)+(B)) + \sum_{i=1}^{N-1} F(A+ixH).$$

ESERCIZIO 4.14: Parametrizzare la funzione INTEG tenuto conto della funzione da integrare. Portare degli esempi di istanziazione.

La procedura potrà essere compilata separatamente:

```
WITH FUNZIONI-MAT; USE FUNZIONI-MAT;
PROCEDURE INTEGRAZIONE IS
     GENERIC
       WITH FUNCTION F(X:FLOAT) RETURN FLOAT;
     FUNCTION INTEG(A,B: IN FLOAT;N: IN INTEGER) RETURN FLOAT
          IS SEPARATE ;
         .
         .
     FUNCTION INTEG-SIN IS NEW INTEG(SIN);
BEGIN
    .
    .
    .
    Z:=INTEG-SIN(0.0,2.0*PI,N=>50);
    .
    .
    .
```

```
END  INTEGRAZIONE;
---------
SEPARATE (INTEGRAZIONE)
FUNCTION  INTEG(A,B: IN  FLOAT;N IN  INTEGER) RETURN  FLOAT  IS
        .
        .
        .
END  INTEG;
```

A parte i task ai quali il Capitolo VI è dedicato, abbiamo visto i mezzi che sono messi a disposizione del programmatore ADA per assicurare la modularità dei programmi.

La grande innovazione è il concetto di package che (abbinato alla compilazione separata), è l'ideale per creare delle biblioteche di risorse.

Il package fornisce dei meccanismi che consentono di nascondere all'utilizzatore il funzionamento interno delle risorse messe a sua disposizione.

Inoltre, abbiamo visto che uno dei concetti principali di queste risorse è la parametrizzazione ottenuta:

— con vincoli che si possono imporre a dei tipi non vincolati,

— assegnazione di valori a dei parametri di sottoprogrammi.
Quest'ultimo concetto è il solo esistente nei linguaggi classici,
— con istanziazione di unità generiche per dei valori dei loro parametri.

Si noterà l'analogia fra la definizione di un tipo derivato:

```
TYPE T IS NEW INTEGER RANGE 1...N;
```

e l'istanziazione di un generico:

```
PACKAGE P IS NEW Q(MISURA=>N);
```

CAPITOLO 5

AMBIENTE STANDARD ED INPUT/OUTPUT

L'AMBIENTE STANDARD

Ogni programma ADA si svolge in un ambiente di dati predefiniti.

Le definizioni di questi dati sono raggruppate in un package chiamato STANDARD e tutto avviene come se il blocco più esterno del vostro programma fosse scritto:

```
WITH STANDARD; USE STANDARD;
PROCEDURE PROGRAMMA-PRINCIPALE IS ...
```

ma questo è automatico: non dovete mai scrivere la prima linea.

Stando così le cose, si può considerare che l'effetto di un'istruzione WITH consiste nell'aggiungere i package indicati alla fine del package STANDARD.

Ecco l'abbozzo della specificazione del package STANDARD.

Rinviamo al manuale di riferimento per una descrizione completa (scriveremo dc per dire "definito dal compilatore").

```
PACKAGE  STANDARD  IS
      TYPE  BOOLEAN  IS (FALSE,TRUE);
      FUNCTION  " NOT "(X:BOOLEAN)  RETURN  BOOLEAN;
      FUNCTION  " AND "(X,Y:BOOLEAN)  RETURN  BOOLEAN;
     -- analogo per " OR " e " XOR "
```

```
TYPE SHORT-INTEGER IS RANGE DC;
TYPE INTEGER IS RANGE DC
-- stessa cosa per LONG-INTEGER
FUNCTION "+"(X:INTEGER) RETURN INTEGER;
-- stessa cosa per "-" e ABS e per gli operatori
-- diadici"+","-","*","/"," REM "," MOD " e
FUNCTION "**"(X:INTEGER,Y:INTEGER RANGE O..INTEGER'LAST)
                RETURN INTEGER;
-- stessi operatori per gli altri due tipi interi
-- stessa cosa par i tre tipi reali:SHORT-FLOAT,
-- FLOAT e LONG-FLOAT:
TYPE FLOAT IS DIGITS dc RANGE dc;
-- specificazione degli operatori per i tre tipi
-- reali: monadici("+","-",ABS)
-- e diadici ("+","-","*",''/" e "*"), per esempio:



FUNCTION "*"(X:FLOAT;Y:INTEGER) RETURN FLOAT;
-- definizione e i caratteri ASCII.
TYPE CHARACTER IS
    (nul,soh,.....' ','****',......
    '0','1',.....'9',':',......
    ''','A','B',.....'Z','****',.......
    'a','b',.....'z','****',del);
-- package che permette di associare un identificatore a
-- ciascun carattere :
PACKAGE ASCII IS
  NUL: CONSTANT CHARACTER:=nul;
      .
      .
      .
  DEL: CONSTANT CHARACTER:=del;
  EXCLAM: CONSTANT CHARACTER:='!';
      .
      .
      .
  LC-A: CONSTANT CHARACTER:='a';
      .
      .
  LC-Z: CONSTANT CHARACTER:='z';
  -- lista completa nell'annesso II.



END ASCII;
-- tipi e sotto-tipi predefiniti
SUBTYPE NATURAL IS INTEGER RANGE 1..INTEGER'LAST;
SUBTYPE PRIORITY IS INTEGER RANGE dc;
TYPE STRING IS ARRAY (NATURAL RANGE <>) OF CHARACTER;
TYPE DURATION IS DELTA dc RANGE dc;
-- le eccezioni (lista all'annessi III) come :
CONSTRAINT-ERROR: EXCEPTION ;
-- il package definito dal compilatore:SYSTEM
PACKAGE SYSTEM IS
   TYPE SYSTEM NAME IS tipo enumerativo dc;
```

```
  NAME: CONSTANT   SYSTEM-NAME:=dc;
  STORAGE-UNIT: CONSTANT :=dc;-- unita' di memoria
  MEMORY-SIZE: CONSTANT :=dc;-- dimensione della memoria
  MIN-INT: CONSTANT :=dc;-- intero minimo
  MAX-INT: CONSTANT :=dc;-- intero massimo
END  SYSTEM;
PRIVATE
   FOR  CHARACTER  USE (0,1,2....126,127);
  -- definisce i codici ASCII senza saltarne nessuno
   PRAGMA  PACK(STRING);
   END  STANDARD;
```

Altri elementi sono predefiniti nel linguaggio, oltre a questo package. Essi sono:

— il package CALENDAR e la procedura generica SHARED-VARIABLE-UPDATE che vengono usate a seconda dei task (cfr. Capitolo VI),

— le procedure generiche UNCHECKED-ALLOCATION (cfr. p. 68) e UNCHECKED-CONVERSION (cfr. p. 152),

— i package di ingresso-uscita che vedremo tra poco INPUT-OUTPUT (che è generico), TEXT-IO e LOW-LEVEL-IO.

I PACKAGE DI INPUT-OUTPUT

La definizione degli input/output in un linguaggio di alto livello colloca l'ideatore fra due fuochi: o non si definisce nulla e si perde la portabilità perché i differenti realizzatori di compilatori definiscono norme differenti (caso di ALGOL), oppure si stabilisce un set di istruzioni (caso di FORTRAN) o delle procedure standard (caso di PASCAL) e si è troppo restrittivi.

ADA ha sufficiente elasticità e metodi di parametrizzazione da consentire di definire dei package di input/ouput che determinano le cose in modo tale da consentire una buona omogeneità fra le differenti realizzazioni, tesi a salvaguardare la possibiltà di soddisfare tutti i casi particolari grazie a dei sovraccarichi o a delle istanziazioni.

IL PACKAGE GENERICO INPUT-OUTPUT

Questo package fornisce dei tipi file e delle operazioni sui file.

Un file è visto come una sequenza illimitata di elementi dello stesso tipo. Questo tipo è il parametro del package. Vi occorrerà dunque istanziare il package per ciascun tipo di dati che volete trattare:

```
PACKAGE E-S-REALI IS NEW INPUT-OUTPUT
(ELEMENTO-TIPO=> FLOAT);
```

Ciascun esemplare del package definisce tre tipi di file:

IN-FILE che non può essere che letto, OUT-FILE che non può essere che scritto, e INOUT-FILE che può essere sia letto che scritto, (su di un file INOUT, l'aggiornamento locale dei record è possibile).

La maggior parte delle procedure del package sono sovraccaricate sui tre tipi. Così, dopo l'istanziazione qui sopra, si potrà scrivere:

```
RISULTATI: E S REALI.OUT FILE;
```

che dichiara RISULTATI come file di reali in sola scrittura.

Se non si hanno che dei reali da trattare, si può anche scrivere

```
USE E-S-REALI; POI,
RISULTATI:OUT-FILE;
```

Un nome come RISULTATI è l'identificatore interno del file. Ogni operazione avrà la forma per esempio:

```
WRITE(RISULTATI,DATO);
```

cioè questo identificatore nel programma servirà a fare riferimento al file.

Il file dispone anche di un **nome esterno** che è una stringa di caratteri che lo identifica nei confronti del sistema di gestione. L'organizzazione di questa stringa di caratteri dipende dal sistema. Certi elementi possono servire a precisare su quale periferica il file si trova ecc... Senza entrare nei particolari, scriveremo questa stringa come: "XXXX".

Un certo numero d'operazioni serve ad associare il file interno con il nome esterno.

Prima di ogni operazione deve essere creato un file. È l'oggetto della procedura CREATE che ha due versioni per i file OUT-FILE e INOUT-FILE. Non c'è versione per IN-FILE poiché la creazione implica una scrittura:

PROCEDURE CREATE(FILE: IN OUT OUT-FILE;NAME: IN STRING);
PROCEDURE CREATE(FILE:IN OUT INOUT-FILE;NAME: IN STRING);

Esempio:

CREATE(FILE=>RISULTATI,NAME=>"@0:ZOZO")

In seguito, prima di ogni accesso, deve essere aperto un file che esiste: è la procedura OPEN che stabilisce la corrispondenza grazie alla quale non appena scriverò sul file risultati,il sistema saprà che si tratta del file "@ 0:ZOZO":

PROCEDURE OPEN(FILE: IN OUT IN-FILE;NAME:IN STRING);

e dei sovraccarichi per OUT-FILE e INOUT-FILE.

Nota: l'operazione CREATE contiene l'apertura del file in scrittura.

Alla fine di tutte le operazioni, si deve sopprimere la corrispondenza cioè effettuare la chiusura del file:

PROCEDURE CLOSE(FILE: IN OUT IN-FILE);

e dei sovraccarichi per OUT FILE e INOUT FILE.

Questa procedura non deve essere confusa con:

PROCEDURE DELETE(NAME: IN STRING)

che sopprime il file esterno interessato:

CLOSE(RISULTATI);

indica che per il momento si smette di utilizzare il file"@0:ZOZO" per il nome RISULTATI. Ma si potrà riaprire il file quando si vorrà, mentre:

DELETE("@ 0:ZOZO");

sopprime fisicamente il file. Non vi si potrà più accedere.

Le funzioni:

FUNCTION IS -OPEN(FILE:IN IN-FILE) RETURN BOOLEAN;

e

FUNCTION NAME(FILE:IN IN-FILE) RETURN STRING;
(sovraccarichi per OUT-FILE e INOUT-FILE)

indicano rispettivamente se un file è aperto e qual è il suo nome esterno.

Le altre operazioni sono gli accessi al file propriamente detti (lettura o scrittura).

Un file in lettura ha una posizione di lettura corrente che è il numero dell'elemento pronto da leggere alla prossima lettura. All'apertura è messa a 1.

Un file in scrittura ha una posizione di scrittura corrente che è il numero del prossimo elemento che si è pronti a scrivere. È messa a 1 all'atto dell'apertura. Queste posizioni sono di tipo intero predefinito FILE-INDEX.

Un file ha una dimensione effettiva che è il numero di elementi scritti ed una posizione di fine che è la posizione dell'ultimo elemento scritto.

Le operazioni possono attivare le eccezioni:

STATUS-ERROR	se si tenta un'operazione su di un file non aperto
USE-ERROR	se si tenta un'operazione incompatibile con le proprietà del file (esempio: WRITE su di un IN-FILE)
DEVICE-ERROR	in caso di malfunzionamento hardware
DATA-ERROR	se il dato trovato è indefinito o incorretto
END-ERROR	se si cerca di leggere oltre la fine del file

Ecco le operazioni:

PROCEDURE READ(FILE: IN IN-FILE;ITEM:OUT ELEMENTO-TIPO);
(sovraccarico per INOUT-FILE)

legge l'elemento ITEM sul file FILE alla posizione corrente e avanza alla

posizione definita successiva.

PROCEDURE WRITE(FILE: IN OUT -FILE;ITEM: IN ELEMENTO-TIPO);
(sovraccarico per INOUT-FILE)

scrive l'elemento ITEM sul file FILE alla posizione corrente, avanza alla posizione seguente e incrementa la dimensione corrente del file.

FUNCTION NEXT-READ(FILE: IN IN-FILE) RETURN FILE-INDEX;
FUNCTION NEXT-WRITE(FILE: IN OUT-FILE) RETURN FILE-INDEX;
(sovraccarichi per INOUT-FILE)

forniscono rispettivamente la posizione corrente di lettura e di scrittura.

PROCEDURE RESET-READ(FILE: IN IN-FILE);
PROCEDURE RESET-WRITE(FILE: IN OUT-FILE);
(sovraccarichi per INOUT-FILE)

riportano a 1 la posizione corrente di lettura o di scrittura, rispettivamente.

Queste due procedure che sono equivalenti al REWIND di FORTRAN consentono il trattamento dei file sequenziali.

PROCEDURE SET-READ(FILE: IN IN-FILE;TO: IN FILE-INDEX);
PROCEDURE SET-WRITE(FILE: IN OUT -FILE;TO: IN FILE-INDEX);
(sovraccarichi per INOUT-FILE)

Queste due procedure fissano al valore indicato da TO la posizione di lettura corrente o la posizione di scrittura corrente.

SET-READ(FILE=> FILE,TO=> 10);
—— posiziona al record numero 10
READ(FILE,ITEM=>DATO);
—— legge DATO sul record numero 10

Queste due procedure consentono di gestire dei file ad accesso casuale (poiché si può dire dove si vuole leggere o scrivere, senza preoccuparsi

dell'ordine), dei file sequenziali (la posizione è definita rispetto all'origine del file). A partire da questa organizzazione, si può creare un sistema di accessi come lo si vuole: casuale o sequenziale indicizzata o altro...

Le funzioni:

<u>FUNCTION</u> SIZE(FILE: IN IN-FILE) <u>RETURN</u> FILE-INDEX;

e

<u>FUNCTION</u> LAST(FILE: IN IN-FILE) <u>RETURN</u> FILE-INDEX;
(sovraccarichi per OUT-FILE e INOUT-FILE)

forniscono rispettivamente la dimensione corrente e la posizione finale del file. Queste due funzioni sono differenti, coincidono solo se — ed è il caso più frequente — si crea il file scrivendo effettivamente tutti i record consecutivamente, cioè senza lasciare dei buchi (record non definiti).

La funzione:

<u>FUNCTION</u> END -OF-FILE(FILE: <u>IN</u> IN-FILE) <u>RETURN</u> BOOLEAN;
(sovraccarico per INOUT-FILE)

restituisce TRUE se la posizione di lettura corrente supera la posizione di fine file.

La procedura:

<u>PROCEDURE</u> TRUNCATE(FILE: <u>IN</u> OUT-FILE;TO: <u>IN</u> FILE-INDEX);
(sovraccarico per INOUT-FILE)

stabilisce la fine del file alla posizione specificata da TO e quindi elimina i record che la seguono. La dimensione corrente è aggiornata.

ESERCIZIO 5.1: Dato un file di reali, aggiungere il numero X alla fine del file.

ESERCIZIO 5.2: Aggiornamento in accesso sequenziale relativo (è più semplice che in sequenziale).

Lo schedario del personale di un'azienda ha la seguente struttura di record:

numero d'impiegato	*(intero)*
nome	*(stringa di 10 caratteri)*
indirizzo	*(stringa di 30 caratteri)*
numero INPS	*(intero di 13 cifre)*
impiego	*(stringa di 10 caratteri)*
grado	*(intero)*

Lo schedario è ordinato secondo il numero crescente di impiegato. In pratica, il numero di impiegato è uguale alla posizione della registrazione nel file.

Si ha anche un file movimento dove le registrazioni hanno la stessa struttura, ma dove l'ordinamento è indifferente. Una registrazione si interpreta sotto la forma: numero d'impiegato, nuovi dati riguardanti questo impiegato.

Effettuare l'aggiornamento.

Ecco ora l'abbozzo del package INPUT-OUTPUT. Per il testo completo, rinviamo al manuale di riferimento.

```
GENERIC
    TYPE  ELEMENT-TYPE  IS   LIMITED   PRIVATE ;
PACKAGE  INPUT- OUTPUT  IS
    TYPE  IN-FILE  IS   LIMITED   PRIVATE ;
    TYPE  OUT-FILE  IS   LIMITED   PRIVATE ;
    TYPE  INOUT-FILE  IS   LIMITED   PRIVATE ;
    TYPE  FILE-INDEX  IS   RANGE  0..dc (definito dal compilatore);
-- le procedure di manipolazione dei file
```

```
-- e i loro sovraccarichi di
PROCEDURE  CREATE ...
-- a
PROCEDURE  TRUNCATE ...;
-- le operazioni di input-output e i loro sovraccarichi da
PROCEDURE  READ ...
-- a
FUNCTION  END-OF-FILE ... ;
-- le eccezioni (elenco appendice  III)
PRIVATE
-- la parte privata
END  INPUT-OUTPUT;
```

IL PACKAGE TEST-IO

Questo package fornisce delle operazioni di input-output con un file esterno presentato sotto forma di caratteri. Il suo scopo è di servire al dialogo input-output con l'uomo, di norma tramite periferiche standard come la tastiera e lo schermo.

Esso fa appello a INPUT-OUTPUT e, quindi, fornisce tutte le possibilità descritte qui sopra. Fornisce inoltre delle procedure GET e PUT che consentono di convertire un oggetto di tipo determinato in stringa di caratteri e di scambiarlo con l'esterno.

Gli input-output testo non sono definiti per i file INOUT. Oltre il loro argomento ITEM(IN per PUT, OUT per GET) che definisce l'oggetto da scambiare, queste procedure hanno un argomento che definisce il file che le riguarda. Se questo argomento è assente (ci sono dei sovraccarichi che lo consentono), si considera che l'operazione concerne il file di input corrente o quello di output corrente. Alla partenza del programma, questi file sono associati al file di input standard e a quello di output standard definiti dal sistema (generalmente una tastiera e uno schermo, oppure un lettore di schede o una stampante).

Infine, l'effetto della scrittura o della lettura tramite TEXT-IO di un carattere ASCII di controllo (oltre ai 95 caratteri stampabili) dipende dal compilatore.

Vedremo le operazioni di manipolazione dei file standard, le operazioni di impaginazione e quelle di lettura/scrittura di oggetti di diversi tipi.

File standard

FUNCTION STANDARD-INPUT RETURN IN-FILE;
fornisce il file di input standard (è il file corrente per default all'inizializzazione).

FUNCTION STANDARD-OUTPUT RETURN OUT-FILE;
fornisce il file di output standard.

FUNCTION CURRENT-INPUT RETURN IN-FILE;

e

FUNCTION CURRENT-OUTPUT RETURN OUT-FILE;
forniscono rispettivamente il file di input e quello di output correnti che sono determinati dalle operazioni ove non è dato un argomento file.

PROCEDURE SET-INPUT(FILE: IN IN-FILE);
PROCEDURE SET-OUTPUT(FILE: IN OUT-FILE);
fissano il file di input corrente o quello di output corrente come se fosse il file che è stato fornito che deve essere aperto.

Impaginazione

Un file-testo è considerato come una sequenza di linee, separate da segni di fine di linea (che non vengono considerati come caratteri del file). Ogni linea è formata da colonne e ogni carattere occupa una colonna. Linee e colonne sono numerate a partire da 1. La lunghezza della linea può essere fissata o no e in momenti diversi della scrittura del file può assumere diversi valori.

Ad ogni istante, per un file testo aperto si conoscono la linea corrente e la colonna corrente che determinano la posizione di partenza del prossimo GET o PUT.

```
FUNCTION COL(FILE: IN IN-FILE) RETURN NATURAL;
FUNCTION COL(FILE: IN OUT-FILE) RETURN NATURAL;
FUNCTION COL RETURN ;-- file di uscita corrente per default
```

fornisce il numero di colonna in cui ci troviamo.

PROCEDURE SET-COL(FILE: IN IN-FILE;TO: IN NATURAL);
(sovraccarichi per OUT-FILE e senza argomento FILE (file di output corrente))

stabilisce il numero di colonne dal valore specificato da TO.

Esempio:

se TAB: ARRAY (1...10)OF NATURAL; è una matrice di posizioni di tabulazione.
SET-COL(TAB(P)); posiziona il file standard alla P-esima posizione di tabulazione

FUNCTION LINE(FILE: IN IN-FILE) RETURN NATURAL;
(sovraccarichi per OUT-FILE e senza argomento FILE(file di output corrente));

fornisce il numero di linea in cui ci si trova.

PROCEDURE NEW-LINE(FILE: IN OUT-FILE;SPACING: IN NATURAL:=1);
(manca il sovraccarico per FILE, (file di output corrente), non c'è IN-FILE (vedere SKIP-LINE))

riporta a 1 il numero di colonna e aggiunge SPACING al numero di linea (SPACING=1 fa andare a capo, 2 fa saltare una linea). Se la larghezza di linea è fissata, aggiunge gli spazi che occorrono.

PROCEDURE SKIP-LINE(FILE:IN IN-FILE;SPACING: IN NATURAL:=1)
(manca il sovraccarico per FILE, (file di input corrente); niente OUT-FILE (vedere NEW-FILE))

riporta a 1 il numero di colonna e aggiunge SPACING al numero di linea, equivale a saltare SPACING-1 linee.

FUNCTION END -OF-LINE(FILE: IN IN-FILE) RETURN BOOLEAN;
FUNCTION END-OF-LINE RETURN BOOLEAN;—— (file d'input corrente) restituisce TRUE se non si hanno più caratteri da leggere sulla linea.

Esempio:

IF END -OF-LINE THEN SKIP-LINE; END IF;
FUNCTION LINE-LENGHT(FILE: IN IN-FILE) RETURN INTEGER;
(mancano sovraccarichi per OUT-FILE e FILE (file di output corrente))
fornisce la larghezza di linea attualmente specificata; restituisce 0 se la larghezza non è fissata.

PROCEDURE SET-LINE-LENGTH(FILE: IN IN-FILE;N: IN INTEGER);
(mancano sovraccarichi per OUT-FILE e FILE (file di output corrente))
fissa la larghezza di linea al valore specificato da N. N=0 vuol dire "non fissato".

Esempio:

SET-LINE-LENGTH(N=>80);

Input-output di caratteri

PROCEDURE GET(FILE: IN IN-FILE;ITEM: OUT CHARACTER);
(manca sovraccarico per FILE (file di input corrente))

restituisce il carattere letto specificato alla posizione ove si trovava nel suo file. Si aggiunge 1 al numero di colonna corrente tranne se il segno di fine di linea precede immediatamente il carattere letto. In questo caso ci si posiziona al primo carattere della linea seguente.

PROCEDURE PUT(FILE: IN OUT-FILE;ITEM: IN CHARACTER);
(manca il sovraccarico per FILE (file di output corrente))

il carattere viene scritto nella posizione attuale del file corrispondente. Si aggiunge 1 al numero di colonna corrente tranne se, fissata la larghezza di linea, si è appena scritto l'ultimo carattere di una linea nel qual caso si scrive un segno di fine linea e ci si posiziona al primo carattere della linea successiva.

Queste procedure sono anche sovraccaricate per ITEM di tipo STRING. Esse devono essere chiamate per una stringa di lunghezza n determinata e sono allora equivalenti a chiamare n volte di seguito la procedura corrispondente per un carattere.

FUNCTION GET-STRING(FILE: IN IN-FILE) RETURN STRING;
(manca il sovraccarico per FILE (file di input corrente))

effettua dei GET successivi saltando gli spazi in testa (spazi, tabulati, segni di fine di linea) e restituisce la sequenza di caratteri trovati fino al prossimo spazio vuoto (escluso).

Esempio:

posizione di partenza

BUONGIORNO SIGNORE

GET_STRING ( ) sarà BUONGIORNO: abbiamo saltato gli spazi fino a B. Ci siamo fermati allo spazio fra BUONGIORNO e SIGNORE.

FUNCTION GET-LINE(FILE: IN IN-FILE) RETURN STRING;
(manca il sovraccarico per FILE (input corrente))

restituisce la successiva sequenza di caratteri trovata fino al prossimo carattere di fine di linea escluso. Si posiziona in seguito al di là di questo fine linea in modo che dei richiami successivi a GET-LINE forniscano le linee successive.

PROCEDURE PUT-LINE(FILE: IN OUT-FILE;ITEM: IN STRING);
(manca il sovraccarico per FILE (output corrente))

scrive la stringa data sul file voluto poi aggiunge un segno di fine di linea.

ESERCIZIO 5.3: ZOZO e ZAZA sono due file-testo. Ricopiare ZOZO su ZAZA conservando la stessa struttura di linee.

Input-output di dati d'altro tipo

Questi dati sono tutti trattati allo stesso modo, un elemento per volta (elemento è qui usato col significato di unità lessicale).

In output, si scrive una stringa di caratteri che obbedisca alla sintassi delle costanti del tipo.

In input, si prende la più lunga stringa di caratteri che obbedisca alla sintassi, cosa che implica che ci si fermi al primo spazio incontrato (dopo l'inizio dei dati);

D'altra parte, si saltano tutti gli spazi nell'intestazione (spazi, tabulazioni, segni di fine di linea). Questo implica che un dato non può essere a cavallo di due linee. Se si incontra una stringa che non obbedisce alla sintassi del tipo, si genera l'eccezione DATA-ERROR.

In output, un parametro WIDTH specifica la larghezza della zona che si trascriverà. Un dato numerico è allineato a destra, un dato di un tipo enumerativo è allineato a sinistra.

Se la larghezza specificata è insufficiente, si utilizza il numero di caratteri che occorre. WIDTH è 0 per default, quindi se il parametro non è specificato, si utilizza proprio ciò che occorre. Se restano abbastanza caratteri sulla linea, il dato è scritto a partire dalla posizione corrente. Se, in caso di larghezza di linea fissa, non ne resta abbastanza, si va all'inizio della linea seguente.

Ogni procedura ha il sovraccarico ove il parametro FILE sia omesso, cosa che corrisponde al file d'input corrente per GET, e all'output corrente per PUT.

Per i **tipi numerici**, si dispone di tre package generici (interni a TEXT-IO). Devono dunque essere istanziati, mentre TEXT-IO non deve esserlo. Per utilizzare una parte non generica di TEXT-IO, è sufficiente un USE TEXT-IO.

Tipi interi

Il package generico INTEGER-IO contiene:

> PROCEDURE GET(FILE: IN IN-FILE;ITEM: OUT NUM);
> (promemoria, sovraccarico con FILE assente)

legge la literal (decimale o della forma base#..#) preceduta da un eventuale segno e la converte in funzione della base.

> PROCEDURE PUT(FILE: IN OUT-FILE; ITEM: IN NUM;WIDTH:
> IN INTEGER:=0;
> BASE: IN INTEGER RANGE 2..16:=10);

esprime il valore di ITEM senza sottolineature, senza 0 in intestazione (ma scrive 0 per il valore 0) e un segno — nell'intestazione se necessario. Se la base è diversa da 10, utilizza la sintassi abituale con #.

ESERCIZIO 5.4:

```
X:=150; Quali sono le stringhe prodotte da :
PUT(X);
PUT(-X,10);
PUT(X,WIDTH=> 15,BASE=> 2);
```

Tipi in virgola mobile

Il package generico FLOAT-IO contiene:

PROCEDURE GET(FILE: IN IN-FILE;ITEM: OUT NUM);
ITEM può avere una base diversa da 10 (sintassi b#xx.xx#Eyy)

mentre per PUT, si ottengono solo dei numeri decimali. Ciò sarà vero anche per i tipi a virgola fissa,

```
PROCEDURE  PUT(FILE: IN  OUT-FILE;ITEM: IN  NUM;
          WIDTH: IN  INTEGER:=0;MANTISSA: IN  INTEGER:=NUM
           DIGITS ;EXPONENT: IN  INTEGER:=2);
```

L'output è obbligatoriamente in decimale. MANTISSA indica quante cifre significative si vogliono, il valore assunto per default è il numero caratteristico del tipo, eventualmente arrotondato. EXPONENT è il numero di cifre dell'esponente.

Tipi in virgola fissa

Il package generico FIXED-IO contiene:

PROCEDURE GET(ITEM: OUT NUM);

A partire da ora, citiamo le procedure nella versione senza argomento FILE. Ma certamente i sovraccarichi che contengono questo argomento esistono. Esso è IN-FILE per GET e OUT-FILE per PUT.

PROCEDURE PUT(ITEM: IN NUM;WIDTH: IN INTEGER:=0;
FRACT: IN INTEGER:=DEFAULT-DECIMALS);

FRACT è il numero di cifre dopo il punto decimale che si desidera; si arrotonda se è più piccolo del numero di cifre conosciuto. Se è più grande, si

aggiungono degli 0 per completare. Il valore per default è dedotto dagli attributi del tipo (vedere la definizione del package).

Tipo booleano

Le procedure seguenti sono direttamente disponibili:

PROCEDURE GET(ITEM: OUT BOOLEAN);

legge una stringa che può essere TRUE o FALSE (lettere maiuscole o minuscole) e attribuisce a ITEM il valore logico corrispondente.

PROCEDURE PUT(ITEM: IN BOOLEAN;WIDTH: IN INTEGER:=0;
LOWER-CASE: IN BOOLEAN:=FALSE);

stampa TRUE o FALSE secondo il valore di ITEM. Si può chiedere la stampa in minuscole specificando LOWER-CASE=>TRUE.

Tipi enumerativi

Ecco un punto in cui ADA va molto più lontano di PASCAL. In PASCAL, si può definire una variabile di tipo GIORNO che può assumere i valori LUNEDÌ,MARTEDÌ..., ma non si può farla leggere alla tastiera sotto forma della stringa di caratteri LUNEDÌ. In ADA, si può, grazie al package generico ENUMERATION-IO che contiene:

PROCEDURE GET(ITEM: OUT ENUM);

che legge un identificatore (nome simbolico di costante di un tipo enumerativo) senza distinguere le lettere maiuscole dalle minuscole, o una literal carattere (carattere fra apostrofi). Se è una delle costanti del tipo, si dà a ITEM il valore corrispondente. Altrimenti, si dà il via all'eccezione DATA-ERROR.

PROCEDURE PUT(ITEM: IN ENUM;WIDTH: IN INTEGER:=0;
LOWER-CASE: IN BOOLEAN:=FALSE);

stampa il simbolo del valore di ITEM nel tipo enumerativo ENUM. Si può chiedere la stampa in minuscole. Se il tipo enumerativo è un insieme di caratteri ('A','B','C',...), il carattere è stampato fra apostrofi: 'A'.

ESERCIZIO 5.5: Far leggere alla tastiera il giorno della settimana. Se l'utilizzatore ha battuto un giorno da lunedì a venerdì, stampare sullo schermo BUONGIORNO. Se è sabato o domenica, stampare BUON WEEKEND. La tastiera e lo schermo saranno considerati come se fossero le periferiche standard.

IL PACKAGE TEXT-IO

Non citiamo le procedure che sono state qui sopra descritte e rinviamo al manuale di riferimento per il testo completo.

```
PACKAGE  TEXT-IO  IS
     PACKAGE  CHARACTER-IO  IS   NEW  INPUT-OUTPUT (CHARACTER);
     TYPE   IN  FILE  IS   NEW  CHARACTER-IO.IN-FILE;
     TYPE  OUT-FILE  IS   NEW  CHARACTER-IO.OUT-FILE;
-- procedure d'ingresso-uscita di caratteri e stringhe
-- package generico per gli interi :

     GENERIC
       TYPE  NUM  IS   RANGE <>;
       WITH   FUNCTION  IMAGE(X:NUM) RETURN  STRING  IS  NUM'IMAGE;
       WITH   FUNCTION  VALUE(X:STRING) RETURN  NUM  IS  NUM'VALUE;

     PACKAGE  INTEGER-IO  IS
      -- le procedure
     END  INTEGER-IO;
 -- analogo in virgola mobile:

     GENERIC
       TYPE  NUM  IS   DIGITS <>;
       .
       .
       PACKAGE  FLOAT-IO  IS  ....
       END  FLOAT-IO;

 -- analogo per i numeri a virgola fissa:
    GENERIC
      TYPE  NUM  IS   DELTA <>;
      .
      .

      PACKAGE  FIXED-IO  IS
       DELTA  IMAGE: CONSTANT  STRING:=IMAGE(NUM' DELTA -INTEGER
                                   (NUM' DELTA ));
      DEFAULT DECIMALS: CONSTANT  INTEGER:=  DELTA  IMAGE'
                                          LENGTH-2;
   -- le procedure
    END  FIXED-IO;
   -- le procedure per i booleani
   -- il package generico per i tipi enumerativi:
```

```
GENERIC
    TYPE  NUM  IS
    WITH   FUNCTION  IMAGE(X:NUM) RETURN  STRING  IS  NUM'IMAGE;
    WITH   FUNCTION  VALUE(X:STRING) RETURN  NUM  IS  NUM'VALUE;

PACKAGE  NUMERAZIONE-IO  IS
   -- le procedure

END  NUMERAZIONE-IO;
-- le procedure di controllo di impaginazione
-- le procedure di manipolazione dei file standard
-- le eccezioni : da NAME ERROR a END ERROR, abbiamo
-- RENAMES :
   NAME ERROR: EXCEPTION   RENAMES  CHARACTER-IO.NAME ERROR;
       .
       .
       .
   LAYOUT ERROR: EXCEPTION ;
END  TEXT-IO;
```

IL PACKAGE LOW-LEVEL-IO

Questo package consente d'inviare delle stringhe di caratteri di controllo ad una periferica fisica. È formato da procedure sovraccaricate per ogni periferica ed il suo corpo può essere redatto in linguaggio macchina mediante l'ausilio del codice istruzione (cfr. Capitolo 8).

Si avrà così:

```
PACKAGE  LOW-LEVEL-IO  IS
    -- dichiarazioni dei tipi di periferiche e di dati
    -- di comando adattati
    -- le procedure :
    PROCEDURE   SEND-CONTROL(DEVICE:tipo  di perif.DATA: IN
OUT  tipo di dato);
    PROCEDURE  RECEIVE-CONTROL(DEVICE:tipo di perif.DATA: IN
OUT  tipo di dato);
END ;
```

Ciò permette un controllo completo sulle periferiche pur restando nel formalismo del linguaggio ADA.

CAPITOLO 6

PARALLELISMO E TASK

ADA offre la possibilità di elaborazioni parallele, cioè consente di suddividere il trattamento in processi che si pensa possano avvenire parallelamente.

Quando si svolgono dei processi paralleli, ci sono delle condizioni di sincronizzazione fra loro ed anche delle condizioni di reciproca esclusione quando essi si dividono dei dati. ADA offre dei mezzi per esprimere queste condizioni.

D'altra parte, ADA non fa alcuna ipotesi né pone condizioni sulla realizzazione pratica del parallelismo. Il programma può essere eseguito senza sfruttare completamente le possibilità di parallelismo, per esempio su di un microprocessore con un compilatore semplice.

Il parallelismo può essere realizzato con dei processori multipli o con divisione di tempo di un solo processore. In ogni caso, il compilatore dovrà assicurare che la semantica dei meccanismi proposti da ADA resti la stessa.

Non possiamo qui esporre in particolare le nozioni che accompagnano il parallelismo, i processi, l'esclusione reciproca. Il lettore ne troverà una descrizione esauriente e chiara in "E.I. Enciclopedia di Elettronica ed Informatica" — Gruppo Editoriale Jackson.

Lo strumento principale di parallelismo in ADA è il task. Un task è una unità di trattamento capace di eseguirsi in parallelo con le altre. Nel corso della sua vita, un task è avviato, eseguito e portato a termine.

Abbiamo visto nel Capitolo I il meccanismo più semplice di sincronizzazione dei compiti, che, malgrado la sua semplicità, può rendere dei servigi. Una unità è terminata solo quando tutti i compiti che ha avviato sono terminati. Ma c'è in ADA un meccanismo più fine, il **rendez-vous** in cui un task ne attende un altro.

Partenza

In ADA preliminare esisteva un'istruzione speciale INITIATE per avviare i compiti. In ADA revisionato, la partenza è automatica. Tutti i compiti dichiarati in un blocco o sotto-programma sono avviati quando si esegue il BEGIN del blocco.

Esecuzione

Quando è attivo, il task può essere sia in esecuzione sia sospeso (in attesa di un rendez-vous).

Ciascun task è diviso in due parti: una specificazione (che dichiara le entry del task) e un corpo che contiene le istruzioni eseguibili. Il corpo del task si presenta sovente come un loop indefinito all'interno del quale si trovano delle scelte fra le differenti operazioni da fare in funzione della sincronizzazione con i compiti esterni.

```
TASK   BODY  nome   IS
    BEGIN
      LOOP
         SELECT
         OR
         OR
         END   SELECT  ;
      END   LOOP  ;
END  nome
```

Termine del task

Un task termina se raggiunge la sua istruzione END, se esegue un'alternativa TERMINATE o se vi è una causa esterna che lo impone (istruzione ABORT o eccezione).

MECCANISMO DEL RENDEZ-VOUS

Un punto di sincronizzazione in un task si chiama un input. Un input è annunciato nella specificazione del task con ENTRY nome(argomenti); ed è materializzata nel corpo del task con un'istruzione:

```
ACCEPT  nome(argomenti) DO
       .
       . -- corpo dell' ACCEPT
   END ;
```

Un'input si comporta come una procedura e, difatti, avrà un'istruzione di chiamata in un altro task.

Distinguiamo quindi il task chiamato e il task chiamante (vedere la figura qui sopra). Si noterà che la chiamata fa riferimento al nome qualificato, poiché non esiste USE per un task (ma si potrebbe ridefinire l'input di task come una procedura mettendo nella parte dichiarativa del corpo di U (prima di BEGIN));

PROCEDURE TE(x :type)RENAMES T.E.;

```
task chiamato                          task chiamante

TASK  T  IS                            TASK  U  IS
   ENTRY  E(X:type);                    .
END ;                                   .
TASK   BODY  T  IS                     END ;
  BEGIN                                TASK   BODY  U  IS
    .                                    BEGIN
    .   1                                 .
    .                                     .
ACCEPT  E(X:type) DO                      .    4
    .                                     .
    .   2                                T.E(X);-- chiamata
    .                                     .
   END ;                                  .    5
    .                                     .
    .   3                              END  U;
    .
END  T;

COLONNA 2
```

Con la partenza del programma sono avviati i due task. Le istruzioni 1 e 4 sono eseguite in parallelo (o successivamente, o alternativamente con segmenti — ADA non lo specifica, e questo non cambia la logica). Si hanno allora due casi che dipendono dal fatto che sia T o U ad arrivare per primo all'ACCEPT.
Il rendez-vous consiste nel fatto che il primo task arrivato all'ACCEPT o alla chiamata corrispondente aspetta l'altro.

Se T arriva per primo all'ACCEPT, è **sospeso** nell'attesa che U arrivi alla chiamata. Se U arriva per primo alla chiamata, è sospeso in attesa che T arrivi all'ACCEPT. È la **sincronizzazione**. Una volta ottenuta la sincroniz-

zazione, cioè quando i due compiti sono arrivati al punto d'incontro, **l'incontro avviene**, questo consiste in:

1 — il task U continua ad essere sospeso, lo sarà fino alla fine del rendez-vous

2 — i parametri (qui X) sono trasmessi come per un sottoprogramma, i parametri IN son trasmessi in fase 2, i parametri OUT sono trasmessi alla fine della fase 3

3 — viene eseguito il corpo dell'ACCEPT (istruzioni 2)

Il fatto che sia eseguito mentre U è sospeso gli dà le proprietà di una **sezione critica** che assicura un'**esclusione reciproca:** le istruzioni 2 possono manipolare dei dati divisi fra i due task (ivi compresi gli argomenti) senza essere disturbati dalle istruzioni di U che modificherebbero questi dati.

Nota: Se la sezione critica si riduce alla trasmissione dei parametri, non vi sono DO ... END; si scrive ACCEPT e (X:tipo);

Quando il rendez-vous è terminato, ognuno dei task continua (o riprende) la sua attività per conto proprio. Le istruzioni 3 e 5 sono eseguite in parallelo.

ESERCIZIO 6.1: Abbiamo il seguente programma:

```
BEGIN
 -- istruzioni indipendenti da X
  .
  . 1
  .
  .
-- istruzioni di calcolo di X indipendenti da 1
  .
  . 2
  .
  .
  X:=...;
-- istruzioni utilizzanti X
  .
  . 3
  .
END  ZOZO;
```

È possibile riscrivere questo programma?

Attributi dei task (o dei tipi task)

T'TERMINATED:	vero booleano se il task T è terminato
T'PRIORITY:	priorità del task T (intero)
T'FAILURE:	eccezione originabile dall'esterno
T'STORAGE-SIZE:	il numero di unità di memoria allocate per l'esecuzione di T

La lista d'attesa di un ACCEPT

Si sottolinea l'asimmetria del rendez vous. Il task chiamante si riferisce al task chiamato mentre il task chiamato non si riferisce al task chiamante. ACCEPT si legge come "accettate una chiamata all'entrata, da cui proviene".

In effetti, si potrebbero avere più task U,V,W che chiamano T.E. Supponiamo che si siano avute queste tre chiamate prima che T arrivi all'ACCEPT. Queste chiamate saranno messe in una lista d'attesa, in ordine di arrivo: (U)T.E/(V)T.E/(W)T.E.

Quando T arriverà all'ACCEPT, il rendez-vous si svolgerà per la prima chiamata della lista (U): essa sarà tolta dalla lista: (V)T.E/(W)T.E ed è allora che T ripasserà sull'istruzione ACCEPT e che la chiamata di V sarà presa in considerazione.

Frattanto, una nuova chiamata di U o una chiamata di un altro task potrà essersi aggiunta alla lista:

(V)T.E/(W)T.E/(U)T.E/(Y)T.E
in corso di soppressione

L'attributo:

E'COUNT

è il numero (intero) di chiamate ad un'entrata di task E in attesa

ESERCIZIO 6.2: Quante chiamate all'entrata T.E provenienti dal task U possono esservi in una lista d'attesa?

ESERCIZIO 6.3: Un semaforo è uno strumento di esclusione reciproca fra più task che "vogliono" accedere ad una certa risorsa. Quando un task vuole accedere alla risorsa, fa SEM.P; accede alla risorsa e libera il semaforo mediante SEM.V (V è l'abbreviazione di VREJGEVEN, liberare in olandese). Scrivete il task SEM.

Istruzione SELECT in un task chiamato

In un task chiamato, l'istruzione SELECT realizza un'attesa selettiva. È della forma:

```
SELECT
     WHEN  condizione 1=>
        alternativa 1;
OR    WHEN  condizione 2=>
        alternativa 2;
OR    WHEN  condizione 3=>
        alternativa 3;
     ......
     ELSE
        sequenza d'istruzioni 4;
END   SELECT ;
```

Le alternative 2,3 ecc... possono essere assenti, come anche la clausola ELSE. Un'alternativa può essere privata della condizione WHEN.

Le alternative hanno la forma:

```
a)  ACCEPT .... DO .... END ;[sequenza delle istruzioni;]
b)  DELAY  temps;[sequenza d'istruzioni;]
c) TERMINATE ;
```

Tramite l'insieme delle alternative, al massimo non può esservi che una forma c. Non si possono avere contemporaneamente una forma b e una forma c. Deve esservi almeno una forma a (con ACCEPT). Se c'è una forma b o una forma c, non si può avere la clausola ELSE.

La selezione avviene in due stadi:

1 — si valutano le condizioni WHEN. Le alternative per le quali la condizione è realizzata (o che non hanno condizione) sono dette aperte. Le altre sono dette chiuse.
Se non c'è alcuna alternativa aperta ed esiste una clausola ELSE, si effettuano le istruzioni corrispondenti e si passa alle istruzioni che seguono l'END SELECT;
Se non c'è alcuna alternativa aperta e non c'è clausola ELSE, scatta l'eccezione SELECT -ERROR.
Se vi sono delle alternative aperte, esse ed esse soltanto sono esaminate nel secondo stadio di selezione.

2 — Si esaminano dapprima le alternative aperte della forma a (con ACCEPT).
Se una di queste può essere eseguita, cioè se l'entrata corrispondente ha almeno una chiamata in attesa, la si esegue, cioè si effettua il rendez vous con la sua sezione critica, poi si esegue la sequenza di istruzioni che segue l'ACCEPT quindi si passa oltre l'END SELECT;
Se nessuna alternativa ACCEPT aperta può dar luogo ad un rendez-vous e se c'è un'alternativa aperta della forma b) dunque con DELAY tempo; si comincia un ciclo di attesa per il tempo indicato (espresso in secondi).
Se durante questo ciclo d'attesa nessuna delle alternative aperte della forma a diviene capace di assicurare un rendez-vous, allo scadere del tempo indicato si esegue la sequenza delle istruzioni che seguono l'istruzione DELAY e si passa oltre l'END SELECT;
Se non vi sono alternative della forma b) e se non c'è TERMINATE, si attenderà indefinitamente fino a quando possa essere effettuato un ACCEPT.
L'alternativa TERMINATE può essere selezionata soltanto se il task considerato dipende da un blocco o da un sottoprogramma e se questo è arrivato alla sua fine, cioè attende che il task sia considerato, per terminare.
Se c'è una clausola ELSE e se nessuna delle alternative aperte (sono tutte della forma a) poichè c'è ELSE) può essere presa, si effettua l'ELSE.

Tutto ciò è abbastanza complesso, ma molto ricco di possibilità. Vediamo qualche esempio.

— accettate l'entrata E a condizione che sia richiesta nei 10 secondi, altrimenti fate scattare l'eccezione TIME-OUT.

```
SELECT
     ACCEPT  E;.....;
  OR
     DELAY  10.0; RAISE  TIME-OUT;
END   SELECT ;
```

```
-  se  la  lista  d'attesa  dell'entry  E e' vuota, accettate
l'entry F; altrimenti, accettate l'entry E;
```

```
SELECT
     WHEN  E'COUNT=0 = >
         ACCEPT  F;...;
     OR
         ACCEPT  E;...;
END   SELECT ;
```

Si vede l'uso dell'attributo COUNT.

ESERCIZIO 6.4: Nell'esempio qui sopra, si può attendere indefinitamente una chiamata. Ora quando il programma è terminato oppure si abbandona il dominio che ha dichiarato il task, non si hanno più chiamate mentre al contrario, la scrittura qui sopra impedisce di terminare. Ponetevi rimedio.

Quesito: Quando ci sono più alternative aperte con ACCEPT che sono già chiamate, quale è scelta?

Ebbene, ADA non impone scelta. La scelta non è determinante e viene lasciata al compilatore. Dunque ogni programma che si basasse su di una scelta particolare sarebbe errato, e, di fatto, se avete dei motivi per fare una scelta anziché un'altra, bisogna esprimere questi motivi nelle condizioni del SELECT.

Ecco ora un esempio più complesso che amministra una buca delle lettere. Un certo numero di task T,U,V... sono suscettibili di trasmettere e di ri-

cevere dei messaggi scambiati con gli altri task. Ogni task T,U,V...conosce il suo "indicativo" ID. Il messaggio è della forma ID-destinatario, ID-mittente,testo. Per trasmettere un MESSAGGIO, il task deve iscriverlo in una variabile MESSAGGIO che serve da buca delle lettere, ma può farlo soltanto se la buca delle lettere è vuota (indicato dal booleano FULL). Un task pronto a ricevere un messaggio lo legge nella cassetta delle lettere. Durante il rendez vous, il task lettore trasmette il suo indicativo e il task gestionale della buca da lettere lo confronta col destinatario del messaggio. Se vi è uguaglianza, il messaggio è considerato come consegnato e la buca delle lettere è considerata come vuota. Il task lettore che ha appena ricevuto un messaggio che gli era destinato tratterà il messaggio.

```
PROCEDURE  CASSETTA  IS
-- dichiarazioni globali:
      TYPE  INDIC  IS (T,U,V,....);
      TYPE  MES  IS
          RECORD
            ID-DEST:INDIC;
            ID-IM   :INDIC;
            TESTO   :STRING(1..50);
          END   RECORD ;


-- il task di gestione della cassetta postale.
      TASK  BAL  IS
         ENTRY  LEGGERE(ID: IN  INDIC;MSG  OUT  MES);
         ENTRY  SCRIVERE(MSG: IN  MES);
      END  BAL;
      TASK   BODY  BAL  IS
         FULL:BOOLEAN:=FALSE;
         MESSAGE:MES;
        BEGIN
          LOOP
            SELECT
                WHEN  FULL =>
                   ACCEPT  LEGGERE(ID: IN  INDIC;MSG: OUT  MES)  DO
                     MSG:=MESSAGGIO;
                      IF  MESSAGGIO.ID-DEST=ID  THEN  FULL:=FALSE;
                      END   IF ;
                   END ;-- DO
             OR
                WHEN   NOT  FULL =>
                   ACCEPT  SCRIVERE(MSG: IN  MES)  DO
                  MESSAGGIO:=MSG; END ;
             OR
                WHEN   NOT  FULL =>
                   TERMINATE ;
             END  SELECT ;
           END   LOOP ;
         END  BAL;
```

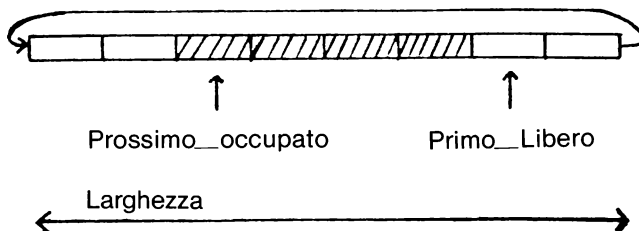
```
-- all'interno del corpo di uno dei task utilizzatori, avremo :

TASK   BODY  T  IS
     ID: CONSTANT  INDIC:=T;
     MSG:MES;
     VOGLIO-LEGGERE,VOGLIO-SCRIVERE:BOOLEANO;
    BEGIN
    .
    .
    .
   MSG.ID-DEST:=U;-- inizializzato diverso da T
    WHILE  VOGLIO-LEGGERE  AND   THEN  MSG.ID-DEST=ID  LOOP
        BAL.LEGGERE(ID,MSG);
    END   LOOP ;-- ciclo sulla lettura finche'
                -- non si e' trovato un messaggio destinato a T
                --trattamento del messaggio ricevuto
    .
    .
    .
    IF  VOGLIO-SCRIVERE  THEN  BAL.SCRIVERE(MSG); END   IF ;
    .
    .
    .
    END  T;
```

Poiché non vi è che un buffer per i messaggi, questo sistema può funzionare convenientemente con n task utilizzatori solo se un task non passa più di 1/n esimo del suo tempo a scambiare dei messaggi.

Ora, tutto il sistema può essere bloccato se c'è un messaggio a) destinato al task T e il task T tarda a venire a leggerlo.

ESERCIZIO 6.5: **Spooling** *(gestione di un buffer circolare).*

Un programma di stampa invia le linee ad un ritmo irregolare; al contrario, la stampante stampa ad un ritmo regolare. Anche se il ritmo medio di invio delle linee non supera la velocità di stampa, possono esservi dei punti compensati da periodi senza invio. Si regolarizza ciò usando un buffer circolare che accumula le linee ricevute.

Esso funziona grazie a due indici PRIMO-LIBERO e PROSSIMO-OCCUPATO. È circolare perché il funzionamento avviene in modulo LARGHEZZA.

Ideate le dichiarazioni globali, il task che invia le linee e il task di stampa; scrivete completamente il task di gestione del buffer.

Si vede che l'istruzione SELECT è estremamente potente e consente di esprimere facilmente ogni algoritmo di simultaneità. Suggeriamo al lettore di esercitarsi a scrivere in ADA gli algoritmi classici come produttore-consumatore, lettore-redattore e algoritmo del banchiere. Il problema principale di questi algoritmi è di evitare gli **interblocchi** (blocchi in cui tutti i task sono sospesi, attendendosi gli uni con gli altri).

Istruzione SELECT in un task o una procedura chiamante

In un task chiamante, l'istruzione SELECT ha due forme:

1 — **chiamata condizionale** della forma:

```
SELECT
    chiamata di un'entry;(sequenza di istruzioni 1);
   ELSE
    sequenza d'istruzioni 2;
END   SELECT ;
```

Si cerca di chiamare l'entrata (di un altro task) indicata. Se il rendez-vous è immediatamente possibile, viene effettuato, così come la sequenza delle istruzioni 1; poi si oltrepassa END SELECT;

Se il rendez-vous non è possibile immediatamente, **non si attende più** (contrariamente alla chiamata semplice). Si effettua la sequenza 2 e si oltrepassa END SELECT;.

Esempio:

```
LOOP
    SELECT
T.E(X);
    ELSE
     -- durante l'attesa fa qualcosa
    END   SELECT ;
END   LOOP ;-- riprova a chiamare
```

2 — chiamata sotto un certo ritardo della forma:

```
SELECT
    chiamata di un'entry;(sequenza 1;)
   OR
     DELAY  tempo;(sequenza 2;)
END   SELECT ;
```

Si cerca di chiamare l'entrata; se il rendez-vous può avere luogo nel tempo indicato, è effettuato poi si esegue la sequenza 1. Se durante il tempo indicato il rendez-vous non ha potuto avere luogo, si esegue la sequenza 2. Dopo la sequenza 1 o 2, si oltrepassa l'END SELECT.

Esempio:

```
SELECT
    T.E(X);
   OR
     DELAY  10.0;
    -- operazioni da fare dopo che la chiamata
    -- non ha potuto essere accettata entro 10 secondi
   END   SELECT ;
```

ESERCIZIO 6.6: Riprendere l'esercizio sullo spooling. Trasformare il task che produce linee in modo che stampi (sul video) un messaggio che segnali che la stampante è sovraccaricata. Si suppone che la stampante funzioni a 1200 l/mn o 20 l/s, quindi si protesta se si deve attendere più di 1/20esimo di secondo.

Istruzione DELAY e il package CALENDAR

L'istruzione:

DELAY espressione;

fa sospendere un task per una durata in secondi uguale al valore dell'espressione.

Una durata è del tipo a virgola mobile predefinito DURATION che ammette almeno il valore 86400 (1 giorno uguale 86400 secondi).

Ciò è in relazione al package predefinito CALENDAR che contiene la funzione CLOCK che fornisce l'ora:

```
PACKAGE  CALENDAR  IS
    TYPE  TIME  IS
        RECORD
          YEAR:INTEGER  RANGE  1901..2099;---  ADA sara'
                                               ancora usato nel 2099?
          MONTH:INTEGER  RANGE  1..12;
          DAY:INTEGER  RANGE  1..31;
          SECOND:DURATION;
        END   RECORD ;
    FUNCTION  CLOCK  RETURN  TIME;
   -- i sovraccarichi delle operazioni per i tipi DURATION e TIME:
    FUNCTION  "+" (A:TIME;B:DURATION) RETURN  TIME;
    FUNCTION  "+" (A:DURATION;B:TIME) RETURN  TIME;
    FUNCTION  "-" (A:TIME;B:DURATION) RETURN  TIME;
    FUNCTION  "-" (A:TIME;B:TIME) RETURN  DURATION;
END  CALENDAR;
```

GENERALIZZAZIONI
FAMIGLIE DI ENTRATE

In un task si può definire una famiglia d'entrate equivalenti allo stesso modo di una matrice indicizzata:

Se R è un intervallo discreto:

```
ENTRY  E(R)(X:tipo;Y:tipo);
ENTRY  CONSOLE-PRINT(1..5)(C: IN  CHARACTER);
```

Si chiama un elemento della famiglia dando un valore all'indice:

T.E(RR) (X=> A;J=>B);

Se si vuole stampare il carattere 'X' sulla quarta console tra le cinque definite qui sopra nel task TELETYPE, si fa:

TELETYPE.CONSOLE-PRINT(4) ('X');

Tipi-task

Ricordiamo l'esercizio 6.3 in cui abbiamo definito un task semaforo SEM per proteggere l'accesso ad una risorsa, per esempio una variabile.

Se avessimo tre variabili: A,B,C da proteggere, bisognerebbe scrivere tre semafori SEMA, SEMB e SEMC e si avrebbe:

```
  SEMA.P;
-- accesso ad A
  SEMA.V
  SEMB.P
-- accesso a B
  SEMB.V
ecc....
```

Ma abbiamo dovuto scrivere tre volte quasi la stessa cosa per definire ciascuno dei semafori.

Possiamo evitarlo definendo un tipo task.

Si scrive:

```
TASK   TYPE  SEM  IS      -- è la stessa notazione
     ENTRY  P;            -- dell'esercizio 6.3 a parte
     ENTRY  V;            -- che si aggiunge la parola TYPE
END ;
TASK   BODY  SEM  IS      -- qui non c'è TYPE
    BEGIN
      LOOP
        ACCEPT  P;
        ACCEPT  V;
      END  LOOP ;
END  SEM;
```

Facendo così, abbiamo definito un **modello** di task, un po' come con un generico.

Per creare un task effettivo SEMA che obbedisca al modello, basta scrivere:

SEMA:SEM;

Dunque, grazie a tre variabili, descriveremo il TYPE SEM come qui sopra, poi:

SEMA, SEMB, SEMC:SEM;

Con un tipo task, si può così creare un dato puntato:

TYPE PTSEM IS ACCESS SEM;
P:PTSEM;

Stando così le cose. l'allocatore:

P:=NEW SEM;

crea e attiva un task semaforo che si può manipolare, per esempio con:

P.P; —— i due P non svolgono lo stesso ruolo!

(il task è designato da P. ALL, per esempio nel P. ALL 'TERMINATED).

Nota: i task svolgono un po' il ruolo di un oggetto LIMITED PRIVATE poiché non c'è per loro né assegnazione né test d'uguaglianza. Il solo modo di trasmettere un task in un altro è di utilizzare dei puntatori:

```
SEMA:=SEMB e' illegale; ma:
PQ:PTSEM;
P:= NEW  SEM;
Q:= NEW  SEM;
P:=Q;       e' legale.
```

COMPLEMENTI

Priorità

Si può assegnare una priorità ad un task con:

PRAGMA PRIORITY (espressione statica intera);

che compare nella specificazione del task. Più l'intero è grande, più il task è urgente.

Ciò non fa che dare un'indicazione al compilatore quando avendo meno processori a sua disposizione in grado di eseguire task eleggibili deve effettuare delle scelte. Sceglierà allora i task con più alta priorità, pur tenendo conto anche di altri criteri, come le risorse richieste.

In caso di uguale priorità, l'ordine non è definito nel linguaggio.

Variabili in comune

I task comunicano normalmente fra loro con scambio di argomenti al momento del rendez-vous.

Se due task agiscono sulla stessa variabile globale, il programmatore deve accertare che i due task non cerchino di modificare questa variabile simultaneamente (da qui l'interesse del task semaforo dell'ESERCIZIO 6.3).

Ma anche per una lettura si pone un problema. Per essere sicuro che si possa sempre accedere alla versione della variabile aggiornata dalle eventuali modificazioni apportate dagli altri task, bisogna accedere alla variabile dopo aver chiamato l'istanziazione conveniente della procedura generica di biblioteca:

```
GENERIC
   TYPE  SHARED  IS   LIMITED   PRIVATE ;
PROCEDURE  SHARED VARIABLE UPDATE(X: IN   OUT  SHARED);
```

Esempio:

```
  X:FLOAT; -- variabile in comune
   PROCEDURE  MAJ  IS   NEW  SHARED VARIABLE UPDATE(FLOAT);
  .
  .
  .
  .
  MAJ(X); -- aggiornamento di X
  Z:=X;   -- accesso ad X
```

Istruzione ABORT

<u>ABORT</u> T,U,V;

forza la chiusura anormale dei task T,U e V. Non dovrebbe essere utilizzato

che in caso estremo. È preferibile, ad esempio, fare un:

```
RAISE T'FAILURE;
```

che lascia a T la possibilità di avere una routine di trattamento della sua eccezione FAILURE e quindi di poter intraprendere un'azione riparatrice.

● ● ●

Lo si è visto, i meccanismi di controllo della simultaneità offerti da ADA sono potenti, ad alto livello e comodi da usare. Grazie ad essi, ADA è capace di avvicinare con successo sia le applicazioni in tempo reale sia la programmazione del software di base.

CAPITOLO 7

ECCEZIONI E TRATTAMENTO DEGLI ERRORI

Le eccezioni sono il mezzo offerto da ADA per il trattamento degli errori o delle situazioni eccezionali. Esse garantiscono **robustezza** ad ADA, consentendo ad un programma di funzionare in presenza di errori. In ADA, un'eccezione ha un nome (identificatore) come ogni oggetto, e segue le regole consuete di visibilità.

C'è tutto un set di eccezioni predefinite nel linguaggio, visibili in tutto il programma.

La loro lista completa è fornita all'appendice III. Citiamo qui:

CONSTRAINT-ERROR : violazione di un vincolo su di un tipo
NUMERIC-ERROR : divisione per 0 o superamento di capacità in un calcolo

o ancora se si utilizzano in modo erroneo le operazioni di input-output

END-ERROR : tentativo di superare la fine del file

L'utilizzatore può ridefinire queste eccezioni (come per ogni oggetto predefinito) o può definire le eccezioni che vuole sotto il nome che vuole.

Ciò viene fatto in una dichiarazione dalla forma:

```
nome[, nome]: EXCEPTION;
```

Esempio:

```
MATRICE-SINGOLARE: EXCEPTION;
```

Bisogna distinguere due avvenimenti a proposito di un'eccezione: l'origine dell'eccezione e il trattamento dell'eccezione.

L'istruzione RAISE

La nascita di un'eccezione può essere sia automatica, sia esplicita.

Le eccezioni predefinite sono originate automaticamente quando le circostanze lo richiedono. Per esempio, a partire da quando un indice di matrice oltrepassa il valore massimo previsto è attivato CONSTRAINT-ERROR. Le eccezioni predefinite possono, e le eccezioni definite dall'utilizzatore devono, essere generate esplicitamente.

La nascita esplicita è fatta da un'istruzione RAISE nome dell'eccezione:

Esempio:

```
IF DETERMINANT=0.0 THEN RAISE MATRICE-SINGOLARE;
RAISE NUMERIC-ERROR;—— eccezione predefinita
```

La forma RAISE; senz'altro è possibile soltanto nella routine di trattamento di un'eccezione, essa riinizializza l'eccezione corrispondente.

Certamente, l'origine di un'eccezione non può aver luogo che nel campo di visibilità di questa eccezione.

TRATTAMENTO DI UN'ECCEZIONE

L'origine di un'eccezione è definita al livello del **blocco** in cui si produce. Blocco è usato in questo capitolo nel senso di blocco (DECLARE BEGIN END), di corpo di sottoprogramma, di package o di task.

Il trattamento di un'eccezione comprende due cose:

1 — l'abbandono del blocco in cui l'eccezione si è originata. Il trattamento si sostituisce alla continuazione dell'esecuzione del blocco. Per un blocco BEGIN .. END si passa alla fine del blocco, per un sottoprogramma si ritorna al punto immediatamente successivo al punto di chiamata di questo sottoprogramma.

2 — l'esecuzione di un certo numero di istruzioni che formano la **routine di trattamento** dell'eccezione (in inglese handler).

Una routine di trattamento si presenta sotto la forma :

WHEN nomi di eccezioni => (stessa sintassi che per CASE)
sequenza di istruzioni;

Esempio:

WHEN NUMERIC-ERROR| MATRICE-SINGOLARE=>
PUT("ERRORE IN TALE PUNTO");

Si può fornire una routine di trattamento anche per un'eccezione predefinita. Comunque, per le eccezioni predefinite, il sistema sa quali azioni intraprendere.

Al posto di un nome d'eccezione si può, come in CASE, specificare OTHERS che significa: "routine di trattamento per tutte le eccezioni per le quali non si sono ancora definite routine".

Esempio:

WHEN OTHERS =>
PUT("AIUTO!");

La sequenza dei WHEN routine d'eccezione è collocata alla fine del blocco, dopo la parola EXCEPTION:

Esempio:

```
PROCEDURE  C  IS
       X,Y: EXCEPTION ;
        BEGIN
         <<A>>....
           I
           I
           I.... ;
      EXCEPTION
        WHEN  X =>
           PUT("ACCIDENTE X");
        WHEN  Y =>
           PUT ("ACCIDENTE Y");
        WHEN   OTHERS  =>
           PUT("AIUTO");
        END  C;
```

Ora, che succede quando si produce l'eccezione?

Rivediamo dapprima il semplice caso corrispondente all'esempio qui sopra. L' "incidente" X ha luogo durante l'istruzione <<A>>; si abbandona la procedura (si saltano le istruzioni I), e si esegue la routine di trattamento di X, poichè questa è presente nel blocco, viene stampata la scritta "INCIDENTE X". In seguito, si ritorna dopo l'istruzione di chiamata di C: se non ci fosse stato l'incidente si sarebbe eseguita completamente la procedura C. Non appena si è verificata l'eccezione immediatamente si è avuta la stampa del messaggio. (In un programma vero, non si avrebbe un tale messaggio ma un'azione tendente a riparare le noie risultanti dall'incidente).

Propagazione delle eccezioni

Quando non c'è routine di trattamento dell'eccezione nel blocco in cui si è verificata l'eccezione, avviene quella che si chiama **propagazione**.

Il blocco in cui l'eccezione si è verificata viene terminato. Poi si risale al blocco che circoscrive immediatamente il primo. Se si tratta di un sottoprogramma, si risale al blocco che chiamava il sottoprogramma. Se nel blocco in cui siamo risaliti c'è una routine di trattamento dell'eccezione, eseguiamo questa routine e poniamo fine al blocco. Se non ce n'é, risaliamo al successivo blocco circoscritto.

Vediamo come la ricerca della routine di trattamento dell'eccezione si propaga per tutti i blocchi annidati o risale la sequenza delle chiamate di sottoprogrammi.

Ne risulta che secondo le circostanze di chiamata, può essere eseguita una diversa routine.

Esempio:

```
PROCEDURE  A  IS
   X: EXCEPTION ;
    PROCEDURE  B  IS
    BEGIN --B
        RAISE  X

    END  B;
    PROCEDURE  C  IS
    BEGIN --C
     ...B; -- chiamata
    EXCEPTION
       WHEN  X=>
```

B fa scattare l'eccezione X ma non fornisce trattamento.

C chiama B e fornisce un trattamento (1).

A chiama C e direttamente anche B. A contiene una routine di trattamento dell'eccezione X(2). Alla chiamata 1, C chiama B. L'eccezione X è fatta scattare. La parte di B che segue l'istruzione che origina X è abbandonata e si risale a C.

```
        trattamento 1;
     END  C
BEGIN --A
       C; -- chiamata 1
       B; -- chiamata 2
EXCEPTION
       WHEN  X=>
       trattamento 2;
END   A;
```

La parte di C che viene dopo la chiamata di B è abbandonata e si esegue il **trattamento 1**, poi si ritorna ad A oltre la chiamata 1.
Durante la chiamata 2 è attivata l'eccezione X. La parte di B che segue l'istruzione che origina X è abbandonata e si esegue il **trattamento 2** poi, se A è il programma principale, il lavoro è terminato, altrimenti, si ritorna nel programma chiamante, oltre l'istruzione che chiamava A.

B fa scattare l'eccezione X ma non fornisce trattamento.
C chiama B e fornisce un trattamento (1).
A chiama C e direttamente anche B. A contiene una routine di trattamento dell'eccezione X(2). Alla chiamata 1, C chiama B. L'eccezione X è fatta scattare. La parte di B che segue l'istruzione che origina X è abbandonata e si risale a C.

La parte di C che viene dopo la chiamata di B è abbandonata e si esegue il **trattamento 1**, poi si ritorna ad A oltre la chiamata 1.

Durante la chiamata 2 è attivata l'eccezione X. La parte di B che segue l'istruzione che origina X è abbandonata e si risale ad A. La parte di A che segue alla chiamata 2 è abbandonata e si esegue il **trattamento 2** poi, se A è il programma principale, il lavoro è terminato, altrimenti, si ritorna nel programma chiamante, oltre l'istruzione che chiamava A.

Queste regole obbediscono alle varianti seguenti:

1 — se l'eccezione si produce durante l'elaborazione della parte dichiarativa di un blocco, si risale subito al blocco circostante (o al blocco chiamante il sottoprogramma).

2 — se l'eccezione si produce durante l'esecuzione delle istruzioni d'inizializzazione del corpo di un package che non fornisce routine di trattamento, l'elaborazione del package è abbandonata e l'eccezione è propagata al blocco che dichiara il package come sottounità. Se è un package di biblioteca, il programma è abbandonato.

3 — se l'eccezione si produce nel corpo di un task che non contiene routine di trattamento, il task è terminato. Non c'è propagazione.

Eccezioni nei task

Quando un task chiama un'entrata in un altro task, viene originata l'eccezione TASKING-ERROR nel task chiamante se il task chiamato termina prima di accettare la chiamata oppure è già terminato al momento della chiamata.

Un rendez-vous fallisce quando:

1 — una eccezione è attivata durante l'ACCEPT e non è trattata localmente. L'eccezione è propagata sia nel blocco contenente l'ACCEPT sia nel task chiamante.

2 — il task contenente l'ACCEPT è terminato in modo anormale (per esempio con ABORT). L'eccezione TASKING-ERROR è attivata nel task chiamante.

Al contrario, la terminazione anormale del task chiamante non ha effetto sul task chiamato. Se il rendez vous non era iniziato, la chiamata è soppressa. Se era cominciato, viene proseguito ed il task chiamato continua.

Ogni task U può far scattare in un altro task T l'eccezione T'FAILURE tramite RAISE T'FAILURE;

Il task che riceve questa eccezione la riceve nel punto in cui si trova sul momento. Se è sospeso in un'istruzione DELAY, l'attesa è soppressa. Se il task stava per chiamarne un altro, la chiamata è soppressa se il rendez-vous non era iniziato. Se lo era, prosegue e in ogni caso il (terzo) task chiamato non ne risente. Se il task ricevitore T è in attesa di un SELECT o di un ACCEPT, l'eccezione è trattata. Se il task T è all'interno di un ACCEPT (nelle istruzioni che seguono il DO dell'ACCEPT), e se non vi è routine di trattamento di T'FAILURE nel corpo di T, allora il rendez- vous è terminato e l'eccezione TASKING-ERROR è fatta scattare nel task chiamante.

Può esservi routine di trattamento di T'FAILURE soltanto nel task (o il tipo di task) T.

ESERCIZIO 7.1: Riprendere il package di gestione di catasta dell'esercizio 4.10 e inserire delle eccezioni per il caso in cui si chiamasse AGGIUNGERE con la catasta piena e TOGLIERE con la catasta vuota.

ESERCIZIO 7.2: Nell'esercizio 4.5, si genera l'eccezione ERRORE-LUNGHEZZA se le dimensioni delle due matrici di cui si vuol fare il prodotto sono non conformi. Suggerire una routine di trattamento. Quale dichiarazione manca?

Soppressione dei test

Le eccezioni predefinite sono originate spontaneamente nelle circostanze volute perchè, ad ogni operazione, il compilatore inserisce dei test per vedere se non sia il caso di sollevare l'eccezione.

Per esempio, ogni volta che utilizzate una variabile indicizzata con l'indice dipendente da una variabile, un test è preparato per verificare che l'indice non superi il suo valore massimo ammesso. Se lo supera, si avrà CONSTRAINT-ERROR.

Si comprende come questi test siano estremamente numerosi e quindi occupano buona parte della durata del programma. ADA offre un PRAGMA che consente di chiedere al compilatore di non effettuare questi test.

Questo PRAGMA è della forma:

PRAGMA SUPPRESS (nome del test)

oppure

PRAGMA SUPPRESS (nome del test, ON=> nome di variabile)

I test possibili sono:

Per CONSTRAINT-ERROR

ACCESS -CHECK: verifica che non si cerchi di accedere ad un oggetto puntato con un puntatore nullo

DISCRIMINANT-CHECK: verifica che una componente di RECORD esista tenuto conto del valore del discriminante

INDEX-CHECK: verifica d'indice

LENGTH-CHECK: verifica del numero d'indici

RANGE-CHECK verifica d'intervallo

Per NUMERIC-ERROR:

DIVISION-CHECK: test di divisione (/, REM o MOD) per zero

OVERFLOW-CHECK: superamento di capacità

Per STORAGE-ERROR: (superamento della dimensione della memoria offerta per l'insieme degli elementi di un tipo puntato al momento di una allocazione):

STORAGE-CHECK

Esempio:

```
PRAGMA SUPPRESS(INDEX-CHECK);—— sopprime tutti i test d'indice
PRAGMA SUPPRESS(INDEX-CHECK,ON=> TAB);—— sopprime i test
                                      —— di indice
                                         unicamente
                                      —— per la matrice
                                         TAB
```

Attenzione, queste opzioni non devono essere utilizzate che durante la ricompilazione di un programma perfettamente a punto. Infatti, le *soppressioni di test non impediscono che gli errori si producano,* impediscono soltanto di controllarne i danni per mezzo del meccanismo delle eccezioni offerto da ADA. Dunque, in caso di errore, quando i test sono soppressi, i risultati possono essere imprevedibili.

CAPITOLO 8

PRAGMI E SPECIFICAZIONI DELLA RAPPRESENTAZIONE

Vediamo in questo capitolo due mezzi offerti da ADA per orientare il lavoro del compilatore.

I **pragmi** sono dei suggerimenti fatti al compilatore che svolgono un ruolo analogo alle direttive del linguaggio assemblatore. C'è un certo numero di pragmi predefiniti nel linguaggio, e un compilatore particolare può includerne altri.

Un dato compilatore ADA può non tener conto di certi pragmi predefiniti se ragioni di complessità o di costo lo giustificano.

Le specificazioni di rappresentazione sono più vincolanti per il compilatore, impongono il modo di rappresentare certi dati o tipi, a volte al livello del bit, cosa che consente, in certe applicazioni, di trarre il massimo vantaggio dalle caratteristiche della macchina utilizzata.

PRAGMI

Ecco i principali pragmi:

PRAGMA CONTROLLED (nome di tipo cui si accede); indica che la zona di memoria di un oggetto del tipo cui si accede non sarà recuperata che uscendo dal campo di validità del tipo cui si accede.

PRAGMA INCLUDE ("nome di file"); fa includere nel testo da compilare, al posto del pragma, il testo trovato nel file di nome indicato.

PRAGMA INLINE (nome di sotto-programma); i sotto-programmi indicati devono essere sviluppati nel testo ad ogni chiamata.

PRAGMA INTERFACE (linguaggio, sottoprogramma); indica che il corpo del sottoprogramma è scritto nel linguaggio indicato, di cui bisogna dunque rispettare le convenzioni. Esempio: PRAGMA INTERFACE(FORTRAN,ATAN).

PRAGMA LIST(ON); indica che si vuole il listing del programma.

PRAGMA LIST(OFF); sospende il listing del programma (ON lo fa riprendere).

PRAGMA OPTIMIZE(TIME O SPACE) specifica che le ottimizzazioni fatte dal compilatore devono piuttosto essere dirette verso un risparmio di tempo (TIME) o di area di memoria (SPACE). Si applica al blocco nella parte dichiarativa nel quale esso si trova.

PRAGMA PRIORITY(n); assegna la priorità n al task nel quale si trova.

PRAGMA SUPPRESS (nome di test, ON=> identificatore); sopprime i test di validità automatica sull'oggetto indicato. La lista dei test possibili è alla fine del Capitolo 7. Il pragma che segue si colloca come le specificazioni di rappresentazione:

PRAGMA PACK(tipo RECORD o ARRAY); specifica che la rappresentazione del tipo indicato deve essere la più compatta possibile. I pragmi seguenti non possono essere che nell'intestazione di una unità di biblioteca. Funzionano a seconda delle rappresentazioni di specificazione:

PRAGMA MEMORY-SIZE(n); fissa a n la dimensione di memoria totale richiesta dall'unità di biblioteca, espressa in unità di memoria.

PRAGMA STORAGE-UNIT(n);fissa a n bit l'unità di memoria. Esempio: PRAGMA STORAGE-UNIT(8);——l'unità—— sarà il byte.

PRAGMA SYSTEM(nome); definisce il nome della macchina. Nome deve essere un simbolo del tipo SYSTEM-NAME definito nel package SYSTEM (vedere Capitolo 5, pag. 101).

SPECIFICAZIONI DI RAPPRESENTAZIONE

Le specificazioni di rappresentazione consentono di specificare la lunghezza di un tipo, la struttura di un record, l'allineamento dei valori di un tipo enumerativo o l'indirizzo di un oggetto.

Esse possono comparire alla fine della parte dichiarativa di un blocco e si applicano agli oggetti dichiarati in questa parte, o in una specificazione di package o di task, specialmente nella parte privata della specificazione di un package. A questo punto, esse si applicano agli oggetti dichiarati nella parte visibile o privata del package, o ad una entrata del task.

In assenza di specificazione di rappresentazione, la rappresentazione è scelta dal compilatore. Se più specificazioni si applicano allo stesso tipo, esse devono essere complementari, per esempio un codice e una lunghezza.

Non è possibile cambiare rappresentazione per un tipo dato. Ma si può definire un tipo derivato e dare una rappresentazione diversa per il tipo derivato.

SPECIFICAZIONE DI LUNGHEZZA

Questa specificazione è della forma:

```
FOR attributo USE espressione;
```

Esempio:

```
FOR INTEGER'SIZE USE 16;
```

che potrebbe essere espresso in modo più chiaro con:

```
BYTE: CONSTANT :=8;
FOR INTEGER'SIZE USE 2*BYTE;
```

La specificazione di SIZE si applica ad ogni tipo oltre che al tipo task, non avendo legami non statici. Essa è espressa in bit.

Si può anche specificare così (e a questo momento la dimensione è espressa in unità di memoria: STORAGE-UNIT):

— la quantità di memoria riservata all'insieme degli oggetti puntati da un tipo access T:
FOR T'STORAGE-SIZE USE...;
— La dimensione di un task o di un tipo di task T:
FOR T'STORAGE-SIZE USE....
— il delta effettivo di un tipo fisso RF:
FOR RF'ACTUAL-DELTA USE valore;

in quest'ultimo caso, valore deve essere reale.

Esempio:

```
TYPE REALE IS NEW FLOAT DELTA 0.001 RANGE —1000..1000;
FOR REALE'ACTUAL-DELTA USE 1.0/2**12;
```

In assenza della specificazione, il sistema avrebbe verosimilmente adottato 1.0/2**10.

C'è un certo numero di attributi dei tipi reali legati alla rappresentazione interna e non ai numeri modello. A pag. 52 ne è citato un certo numero. La loro utilizzazione consente di sfruttare delle prorietà più forti di quelle del tipo che si è specificato, ma ciò può essere a detrimento della portabilità.

Esercizio 8.1: Si dichiara:

```
TYPE REEL IS DELTA 0.01 RANGE —100.0..100.0;
FOR REEL'SIZE USE 12;
```

Che cosa c'è che non va?

Codice di un tipo enumerativo:

Dato un tipo enumerativo, si associa a ciascuno dei suoi simboli un numero intero. I numeri specificati devono seguire l'ordine dell'enumerazione ma non hanno bisogno di essere consecutivi.

Il codice è specificato come segue:

```
FOR tipo enumerativo USE aggregato;
```

l'aggregato dà la lista degli interi con la presentazione abituale.

Esempio:

con il celebre verso di Rimbaud (modificato):

Un nero, due blu, sei rosso...

si scriverebbe:

```
TYPE COLORE IS (NERO,BLU,ROSSO);
FOR COLORE USE (NERO=>1,BLU=>2,ROSSO=>6);
```

Nota: Anche se gli interi non sono consecutivi, gli attributi come SUCC e PRED sono definiti. Per trovare la rappresentazione di SUCC(X), il sistema si occupa di aggiungere la differenza. Il solo inconveniente è che, per esempio per casi in cui il tipo enumerativo serve da indice, i calcoli saranno più lunghi.

Questa proprietà consente di gestire facilmente un linguaggio assemblatore. Si definisce un tipo enumerativo i cui simboli sono i mnemonici delle istruzioni e dove la rappresentazione sarà formata dai codici operativi corrispondenti.

Esempio:

Ecco un estratto della tavola che definisce il linguaggio macchina del microprocessore 6502 (codici in esadecimale).

LDA	STA	CMP	ADC	SBC	BIT	JMP	BEO	BNE	CLC	SEC	JSR	RTS	BOC	BCS
AD	8D	CD	6D	ED	2C	4C	F0	D0	18	38	20	60	90	B0

Si scriverà:

```
TYPE  INSTRUCT  IS (CLC,JSR,BIT,SEC,JMP,RTS,ADC,STA,BCC,LD,
BCS,CMP,BNE,SBC,BEQ);
FOR  INSTRUCT  USE (CLC=> 16#18#,JSR= >16#20#,BIT=> 16#2C#,...
                                ... BEQ=> 16#F0#);
```

Per avere il codice K di un'istruzione I, bisogna fare:

```
I:INSTRUC;
K:INTEGER;
```

```
FUNCTION  CODOP  IS  NEW  UNCHECKED CONVERSION(INSTRUC,INTEGER);
K:=CODOP(I);
```

Per disassemblare l'istruzione di codice K, si fa:

```
FUNCTION  MNEMO  IS  NEW  UNCHECKED-CONVERSION (INTEGER,INSTRUC);
I:=MNEMO(K);
```

Non siamo lontani dall'avere scritto — senza colpo ferire — un assemblatore e un dissemblatore!

Esercizio suggerito 8.2: Trattate completamente, in questo modo, un linguaggio macchina di vostra scelta.

SPECIFICAZIONE DELLA STRUTTURA DI UN TIPO RECORD

La rappresentazione di un record è specificata sotto la forma:

```
FOR  type record  USE
   RECORD  [clausola di allineamento;]
      nome di componente    specificazione di posizionamento
         .
         .
         .
   END  RECORD
```

La eventuale clausola d'allineamento obbliga il record ad essere ad un indirizzo multiplo di una certa misura. Per esempio, si può imporre di cominciare ad un byte pari.

Essa è della forma:

AT MOD espressione statica;

Esempio:

se STORAGE-UNIT è un byte, per cominciare ad un byte pari, si scriverebbe:

AT MOD 2;

per ogni componente, la specificazione è della forma:

nome di componente AT espressione statica semplice RANGE intervallo.

L'espressione indica l'indirizzo relativo all'inizio del record dove comincia questa componente. (L'indirizzo è espresso in unità di memoria, l'inizio di record è a zero).

L'espressione indica l'indirizzo relativo all'inizio del record dove comincia questa componente. (L'indirizzo è espresso in unità di memoria, l'inizio di record è a zero).
L'intervallo descrive da quale bit a quale bit si trova la componente. Il primo bit ha il numero 0.

Non tutte le componenti possono essere specificate. Per quelle che non sono specificate il compilatore sceglie la sua rappresentazione.

Le zone non devono sovrapporsi. Se vi sono delle varianti, le diverse varianti possono essere sovrapposte perché si sostituiscono le une con le altre.

Esempio:

Il PIA 6520/6820 è un circuito d'interfaccia che ha le proprietà seguenti (in breve):

È formato da quattro registri, ciascuno su di un byte

indirizzo	0:	PA	registro dati o direzione della porta A
indirizzo	1:	CRA	registro comando della porta A formato dai bit:
	0:	CA1S	definisce se il CA1 è sensibile ad un impulso positivo
	1:	CA1I	dice se l'impulso su CA1 può creare un'interruzione
	2:	DDA	dice se si accede ad un dato o alla direzione della porta A

3,4, 5:	CA2CTL	definisce il comportamento di CA2
6:	IRQA2	interruzione dovuta a CA2
7:	IRQ1	interruzione dovuta a CA1
indirizzo 2:	PB	registro dati o direzione della porta B
indirizzo 3:	CRB	registro di comando della porta B avente la stessa struttura di CRA.

Si può allora definire:

```
( PRAGMA  STORAGE UNIT(8);)
BYTE: CONSTANT :=1; -- byte.

TYPE  PIA  IS
    RECORD
      PA: ARRAY (0..7) OF  BOOLEAN;
      CA1S   :BOOLEAN;
      CA1I   :BOOLEAN;
      DDA    :BOOLEAN;
      CA2CTL:INTEGER  RANGE  0..7;
      IRQA1 :BOOLEAN;
      IRQA2 :BOOLEAN;
      PB: ARRAY (0..7) OF  BOOLEAN;
  --  CB1S,CB1I,DDB,CB2CTL,IRQB1,IRQB2 analoghi a A
     END   RECORD ;
poi :

   FOR  PIA  USE
      RECORD  -- nessun allineamento utile qui
        PA         AT  0*BYTE  RANGE  0..7;
        CA1S       AT  1*BYTE  RANGE  0..0;
        CA1I       AT  1*BYTE  RANGE  1..1;
        DDA        AT  1*BYTE  RANGE  2..2;
        CA2CTL     AT  1*BYTE  RANGE  3..5;
        IRQA2      AT  1*BYTE  RANGE  6..6;
        IRQA1      AT  1*BYTE  RANGE  7..7;
        PB         AT  2*BYTE  RANGE  0..7;
    --  da CB1S a IRQB1 analoghi, ma  AT  3*BYTE.
      END   RECORD ;
```

Purché ci sia, ma è ciò che il sistema fa automaticamente nella maggior parte dei casi:

FOR BOOLEAN USE (FALSE=>0, TRUE=>1);

si potrà molto facilmente manipolare il PIA.

ESERCIZIO 8.3: Il registro di stato del microprocessore 6502 ha la seguente struttura:

7	6	5	4	3	2	1	0
N	V		B	D	I	Z	C

(il bit 5 è inutilizzato). Scrivere le specificazioni di una copia in memoria di questo registro.

Esercizio suggerito 8.4: Esaminate una scheda tecnica del MOS Technology 6522 (VIA, chip che è una versione perfezionata del PIA) e scrivete le sue specificazioni in ADA.

SPECIFICAZIONI DI INDIRIZZO

Si può specificare l'indirizzo di inizio di un oggetto o di un programma con:

FOR nome USE AT espressione statica semplice;

L'indirizzo di inizio di un sottoprogramma, di un package di un task è l'indirizzo di inizio ove sarà introdotto il codice macchina generato dal corpo di questo blocco.

Due sottoprogrammi diversi non possono essere inseriti allo stesso indirizzo; si potrebbe credere che sia consentita una sovrapposizione con il secondo sottoprogramma che si carica sopra il primo per economizzare la memoria ma non è così; è necessario un compilatore dedicato che possieda dei pragmi e degli attributi per consentire la sovrapposizione.

Esempio:

```
FOR  CHRGET  USE   AT  16#0070#;-- routine collocata in 70 esa
FOR  KEYBDBUFINDEX  USE   AT  16#9E#;-- variabile collocata in 9E
```

ESERCIZIO 8.5: Si hanno due PIA analoghi a quelli del paragrafo precedente collocati in E810 ed E820 esadecimale. Aggiungete le dichiarazioni necessarie.

Una volta fatto ciò, i PIA sono estremamente facili da manipolare. Per autorizzare le interruzioni di CA1 del PIA1, basta scrivere:

PIA1.CA1I:=TRUE;

e le altre manipolazioni in modo analogo.

Pur essendo ADA un linguaggio molto evoluto, consente delle manipolazioni estremamente facili a livello macchina, e quindi può essere un eccellente linguaggio per delle applicazioni di controllo di procedimenti complessi. Inoltre, la specificazione del tipo RECORD è un eccellente strumento di **descrizione** delle caratteristiche hardware come quelle di un PIA.

Interruzioni

Quando è specificato l'indirizzo per l'entrata di un task, esso associa questa entrata ad un'interruzione (il cui indirizzo è l'inizio della routine di trattamento):

```
TASK  INTERRUPTION  IS
   ENTRY  NMI;
   ENTRY  IRQ;
   FOR  NMI  USE   AT  16#1000#;
   FOR  IRQ  USE   AT  16#2000#;
END ;
```

ESERCIZIO 8.6: In BASIC, si dispone dell'istruzione POKE per scrivere il dato che si vuole all'indirizzo voluto; come fare in ADA ?

Introduzione segmenti di programma in linguaggio macchina

Si definisce un package che specifica le differenti istruzioni e i loro formati (modi di indirizzamento), necessariamente legati all'hardware: per esempio, per il 6502, si avrà:

```
PACKAGE  MACH 6502  IS
    TYPE  INDIRIZZAMENTO IMMEDIATO  IS  (...LDA,ADC,... le istruzioni
                              qui ammettono indirizzamento immediato ...);
    TYPE  INDIRIZZAMENTO ASSOLUTO  IS  (.....);
       .
       . stessa cosa per tutti i modi di indirizzamento
       .
-- cosi', verra' segnalato errore se si cerca di utilizzare
-- con un certo modo di indirizzamento un'istruzione che non
-- ammette questo modo.
    TYPE  FORMATO IMMEDIATO  IS
    RECORD
       INDIRIZZAMENTO IMMEDIATO;
       FATTORE:INTEGER  RANGE  0..255;
    END   RECORD ;
   .
   .stessa cosa per gli altri formati per ogni
   .modo di indirizzamento
```

Si hanno le specificazioni di rappresentazione dei mnemonici:

```
     FOR  INDIRIZZAMENTO IMMEDIATO  USE (....LDA=>16#A9#,....);
     --   per ogni modo di indirizzamento, poi per ogni formato:
     FOR  FORMATO IMMEDIATO  USE
        RECORD
           CODICE  AT  0*BYTE  RANGE  0..7;
           FATTORE  AT  1*BYTE  RANGE  0..7;
        END   RECORD ;
         .
         .
         .
         .
    -- il package puo' contenere inoltre le definizioni dei
    -- registri interni, ecc...
END  MACH 6502;
```

In seguito si può costituire una procedura INLINE la cui parte dichiarativa non comporterà che delle clausole USE e dei PRAGMA ed il cui corpo non conterrà che delle istruzioni di codice macchina:

Esempio:

```
PROCEDURE  LINGUAGGIO MACCHINA  IS
    PRAGMA  INLINE(LINGUAGGIO-MACCHINA);
    USE  MACH 6502;
BEGIN
   FORMATO IMMEDIATO'(CODICE=>LDA,FATTORE=>16#FF#);--LDA#$FF
   FORMATO ASSOLUTO'(CODICE=>STA,INDIRIZZO=>16#1000#);--STA $1000
    .
    .
    .
    .
END ;
```

Concludendo, ci si serve di ADA come assemblatore. D'altra parte, per chiamare una routine esteriore in assemblatore, si ha il PRAGMA INTERFACE.

CAPITOLO 9

CONCLUSIONE

Eccoci al termine della panoramica sul linguaggio ADA.

Abbiamo potuto constatare che ADA è notevole sotto più aspetti, ed in particolare per:

— **il carattere completo e universale:** ADA è dotato di particolarità che lo rendono ad un tempo adatto ai trattamenti numerici (e alla costruzione di biblioteche matematiche), ai trattamenti di gestione (eccellenti trattamenti di stringhe di caratteri e operazioni di input-output molto flessibili), e alla programmazione di software di base grazie ai suoi trattamenti in multitasking.

— **la coerenza:** le caratteristiche di ADA non sono tutte originali, sono ispirate ad altre ricerche che gli ideatori hanno voluto incorporare in ADA per farne un linguaggio che rifletta lo stato attuale della programmazione.
Ma, così facendo, hanno compiuto un notevole lavoro di uniformazione dei concetti. ADA si basa su di un piccolo numero di concetti chiave: i tipi, i package, i generici, la parametrizzazione.

— **l'elasticità**: la parametrizzazione è uno dei concetti principali in ADA. Intervenendo a tutti i livelli, gli conferisce una notevole elasticità.

— **il rigore**: ADA è molto seducente per lo spirito in ragione del suo rigore. Questo rigore ha anche come risultato quello di rendere la normativa ADA molto precisa e quindi dovrebbe dotarlo di una portabilità perfetta.

La ricchezza di ADA si accompagna ad un difetto che non è che il riflesso delle sue qualità, ma resta pur tuttavia un difetto; *ADA non può che essere un linguaggio complesso.*

Anche se tutti i formalismi sono stati notevolmente unificati e se certe complicazioni della versione preliminare sono state soppresse nella versione revisionata, ADA resta un linguaggio complesso.

È consolante che questa complessità non sia gratuita e corrisponda in quasi tutti i casi ad una grande potenza di espressione, ma questa complessità esiste. Non si può avere un linguaggio capace di adattarsi a tutti i problemi di analisi numerica senza una certa complessità al livello di trattamento dei numeri reali. Non si può avere un linguaggio capace di trattamenti multitasking senza la corrispondente complessità, anche se questo non è motivo per privarsi del trattamento dei task. E si potrebbe dire la stessa cosa per le altre possibilità di ADA.

Anche se si riprende l'argomento che era quello dei difensori del PL/1 ("sebbene il linguaggio sia complesso, lo si può studiare progressivamente, poco alla volta"), non pensiamo che ADA possa essere il primo linguaggio di programmazione studiato da un principiante in informatica.

Bisogna allora iniziare col BASIC, il linguaggio più semplice da imparare, dunque che non rischia di creare blocchi psicologici contro il computer? Ma il BASIC è forse un po' troppo semplice e non essendo strutturato non incoraggia alla programmazione rigorosa.

Bisogna allora cominciare con PASCAL che è strutturato, e introduce la stessa problematica dei tipi di ADA? PASCAL è però già abbastanza complesso ed ha delle versioni contraddittorie, dunque è capace di disorientare.

Noi consiglieremmo di iniziare col nuovo linguaggio semplice che si sta attualmente introducendo: COMAL. COMAL è — in ultima analisi — un BASIC strutturato. Conserva la semplicità del BASIC pur essendo strutturato, cosa che educa subito alle buone abitudini. Lo si può anche considerare come un PASCAL senza i tipi. Una volta familiarizzato col computer grazie a COMAL(1), lo studente potrà accostarsi ad ADA, con, eventualmente, una tappa intermedia utilizzando PASCAL.

---

(1) COMAL presenta certi particolari molto simili ad ADA, specialmente la struttura IF con una clausola ELSIF (che COMAL scrive ELIF, ma ciò non cambia nulla), e la parola-chiave WHEN nei CASE.

APPENDICE 1

PAROLE-CHIAVE RISERVATE

PAROLA	RUOLO	PAG.
ABORT	annullamento del task	136
ACCEPT	rendez-vous del task	121
ACCESS	definisce un tipo puntatore	65
ALL	ogni elemento puntato da un puntatore	65
AND	operatore logico e dichiaratore di matrice	31
ARRAY	dichiaratore di matrice	54
AT	specificazione d'indirizzo o d'allineamento	152
BEGIN	inizio della parte esecutiva di un blocco, di un sottoprogramma, di un package, di un task	21
BODY	corpo di un package, di un task...	81,121
CASE	struttura caso o variante di RECORD	41,60
CONSTANT	introduzione di una costante	41
DECLARE	inizio della parte dichiarativa di un blocco	21
DELAY	genera un ritardo	132
DELTA	scarto di un tipo reale a virgola fissa	48
DIGITS	numero di cifre di un tipo a virgola mobile	48
DO	inizio della parte critica di un rendez-vous	121
ELSE	alternativa di un IF o SELECT	33,132
	accompagna OR in OR ELSE	32
ELSIF	altrimenti se in un IF	33
END	fine del blocco, del sotto-programma, o della struttura	21
ENTRY	entrata di task	121
EXCEPTION	dichiaratore d'eccezione	139
	inizio delle routine d'eccezione	142
EXIT	uscita dal ciclo	36
FOR	inizio di ciclo su indice corrente	38

PAROLA	RUOLO	PAG.
	specificazione di rappresentazione	149
FUNCTION	dichiaratore di funzione	71
GENERIC	introduce i parametri generici	93
GOTO	trasferimento incondizionato	33
IF	struttura se	33
IN	operatore di appartenenza	30
	intervallo di un FOR	37
	tipo di argomento	72
IS	introduce una descrizione	42,71
LIMITED	tipo privato limitato	81
LOOP	ciclo (FOR, WHILE o indefinito)	36
MOD	modulo	29
	clausola d'allineamento	152
NEW	introduce un nuovo tipo derivato	45
	istanziazione di un generico	94
	allocatore di variabile puntata	67
NOT	operatore di negazione	31
NULL	istruzione nulla o elemento nullo	41,62
	puntatore verso nulla	67
OF	tipo di base di una matrice	54
OR	operatore logico o, alternativa in SELECT	32,131
OTHERS	altri elementi di un aggregato	26
	altre ipotesi in CASE o EXCEPTION	40,141
OUT	tipo di argomento	72
PACKAGE	package	81
PRAGMA	introduce una direttiva	139
PRIVATE	tipo privato	81
	introduce la parte privata di un package	82
PROCEDURE	dichiaratore di procedura	72
RAISE	origina un'eccezione	139
RANGE	gamma di valori	46
	posizione di bit in una parola	152
RECORD	registrazione	60
REM	resto di divisione	28
RENAMES	rinomina, ridefinisce	89

PAROLA	RUOLO	PAG.
RETURN	ritorno da funzione	72
REVERSE	inverte un intervallo	38
SELECT	selezione d'azioni in un task	131
SEPARATE	compilazione separata	91
SUBTYPE	dichiarazione di sottotipo	41
TASK	dichiaratore di task	122
TERMINATE	termina un task	131
THEN	alternativa a IF	33
	accompagna AND in AND THEN	31
TYPE	dichiarazione di tipo	41
USE	utilizzazione di un package	89
	specificazione di rappresentazione	149
WHEN	condizione in EXIT o SELECT	37,131
	ipotesi in CASE o EXCEPTION	40,141
WHILE	ciclo fintanto che	37
WITH	introduce un package di biblioteca	89
	introduce un argomento sotto-programma in un generico	96
XOR	operatore logico o esclusivo	31

NB: Rispetto ad ADA preliminare, sono stati aggiunti REM, WITH e TERMINATE, sono stati eliminati ASSERT, INITIATE e PACKING, RESTRICTED è stato sostituito da LIMITED.

CARATTERI SPECIALI E COMBINAZIONI AVENTI UN SENSO IN ADA

	RUOLO	PAG.
+	addizione	28
—	sottrazione	28
*	moltiplicazione	28
/	divisione	28
&	concatenazione	30
<	inferiore	30
>	superiore	30
=	uguale	30
	nome qualificato	23
	dichiarazione di variabile	41
	identificatore di blocco	35
#	numero di base <> 10	24
E	esponente	24
'	apostrofo:	24
	delimita un carattere isolato	
	attributo	26
	nome qualificato	54
"	virgolette:	
	stringa di caratteri	24
,	virgola	44
	separatore in una lista	
;	terminatore d'istruzione	2
	separa due argomenti in una specificazione	72
(....)	parentesi:	
	aggregato	26
	indice	25
	argomenti	72
|	barra verticale:	
	separa due elementi in una scelta	39
——	introduzione di commenti	3
**	esponenziazione	28

	RUOLO	PAG.
<=	inferiore o uguale	30
>=	superiore o uguale	30
/=	indifferente	30
<<	inizio d'etichetta	33
>>	fine d'etichetta	33
=>	associazione di parametri o di scelta	40,75
< >	("box"): intervallo indefinito	54
:=	assegnazione o inizializzazione	25

APPENDICE 2

I CARATTERI ADA

I caratteri ADA sono definiti dal tipo CHARACTER che enumera i 128 caratteri ASCII. I caratteri non stampabili (caratteri di controllo) sono espressi sotto la forma di un simbolo:

```
TYPE CHARACTER IS
        (nul, soh, stc,....,us,
        ' ','!','"',#','=','%' ....
        ' ','A',....'Z','[', ....
        ..'a','b',...'z',....'}','~',del);
```

Il package ASCII ha il fine di dare un nome esprimibile nel set di caratteri ridotto, a tutti i caratteri.

```
 PACKAGE  ASCII  IS
-- caratteri di controllo
   NUL: CONSTANT  CHARACTER:=nul;
   SOH: CONSTANT  CHARACTER:=soh;
    .
    .
    .
   US : CONSTANT  CHARACTER:=us;
   DEL: CONSTANT  CHARACTER:=del;
-- i caratteri speciali ricevono un nome:
EXCLAM: CONSTANT  CHARACTER:='!'
SHARP: CONSTANT  CHARACTER:='#'
   .
   .
   .
-- vedere tabella qui sopra
-- le lettere minuscole ricevono un nome della forma LC ...
-- (LOWER CASE)
LC A: CONSTANT  CHARACTER:='a';
  .
  .
LC Z: CONSTANT  CHARACTER:='z';
END  ASCII;
```

TABELLA DEI CARATTERI SPECIALI

<table>
<tr><td>EXCLAM</td><td>!</td><td>L_BRACKET</td><td>[</td><td>L_BRACE</td><td>{</td></tr>
<tr><td>SHARP</td><td>#</td><td>BACK_SLASH</td><td>\</td><td>BAR</td><td>|</td></tr>
<tr><td>DOLLAR</td><td>$</td><td>R_BRACKET</td><td>]</td><td>R_BRACE</td><td>}</td></tr>
<tr><td>QUERY</td><td>?</td><td>CIRCUMFLEX</td><td>^</td><td>TILDE</td><td>~</td></tr>
<tr><td>AT_SIGN</td><td>@</td><td>GRAVE</td><td>`</td><td></td><td></td></tr>
</table>

I caratteri di controllo più importanti sono BS (back-space), HT (horizontal tabulation), LF (line feed), VT (vertical tabulation), FF (form feed), e soprattutto CR (carriage return).

APPENDICE 3

ELEMENTI PREDEFINITI

Gli attributi predefiniti sono stati citati ai capitoli 3 e 4.
I pragmi predefiniti sono stati citati al capitolo 8.
I package STANDARD e SYSTEM sono descritti al capitolo 7.
I test di verifica sono stati listati al capitolo 7.

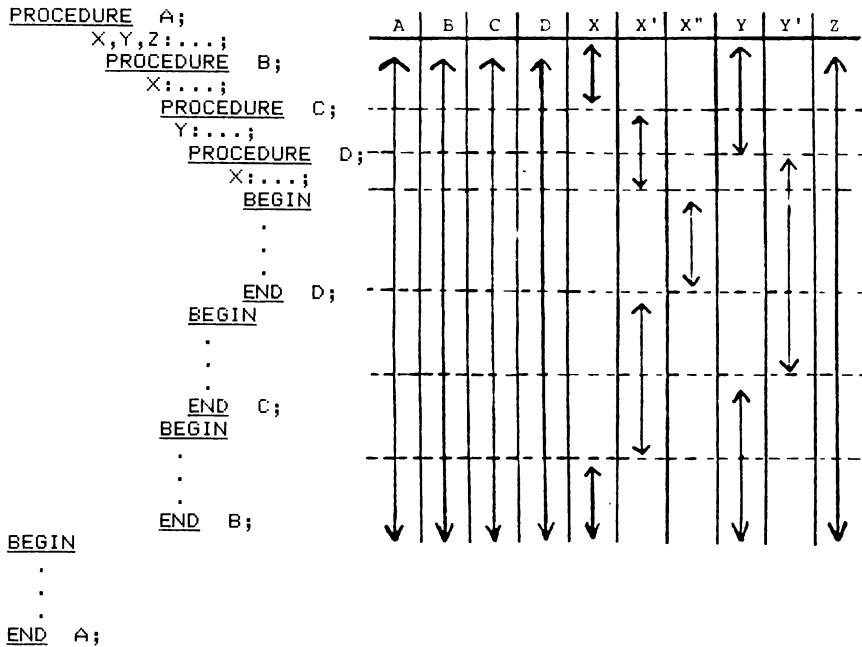
Le eccezioni predefinite generali sono:

CONSTRAINT-ERROR
NUMERIC-ERROR
SELECT-ERROR
STORAGE-ERROR
TASKING-ERROR

quelle che sono predefinite nel package INPUT-OUTPUT e TEXT-IO sono:

NAME-ERROR	(uso di un nome errato)
USE-ERROR	(tentativo di operazione impossibile su di un file)
STATUS-ERROR	(prova a ricreare un file esistente o di utilizzare un file inesistente)
DATA-ERROR	(dato errato)
DEVICE-ERROR	(malfunzionamento hardware di una periferica)
END-ERROR	(cerca di leggere oltre la fine del file)
LAYOUT-ERROR	(cattiva impaginazione: tenta di superare la larghezza della linea)

APPENDICE 4

SOLUZIONE DEGLI ESERCIZI

ESERCIZIO 2.2:

2#1001-1010#
23 (12+11)

ESERCIZIO 2.3:

si può assegnare separatamente ogni elemento del record:

D.GIORNO:=15;
D.MESE:=AGO;
D.ANNO:=1981;

ESERCIZIO 2.5:

Si valuta dapprima 1<5 che dà TRUE e 2>3 che dà FALSE. Resta TRUE<FALSE, cosa che è FALSE, poiché nella definizione del tipo booleano, si è dato FALSE prima di TRUE (TYPE BOOLEAN IS (FAL-SE,TRUE)).

La seconda espressione non è corretta: (5<2) è booleana.

Non si saprebbe calcolare 1 (intero) < qualcosa di booleano.

ESERCIZIO 2.6:

```
(A  AND ( NOT B)) OR (B  AND ( NOT  A)) o
(A  OR  B) AND ( NOT  A  AND  B) o ancora
NOT  A=B (ricordate A e B sono booleani)
```

ESERCIZIO 2.7:

X IN 3...10

ESERCIZIO 2.8:

Siccome A, B e C sono booleani, si può esprimere la condizione sotto la forma:

A AND (NOT (B AND C))

Si passa in seguito all'istruzione 4.

ESERCIZIO 2.9:

```
1)  IF  A=0  THEN
          IF  B=0  THEN
                PUT("INDETERMINATO");
               ELSE
                PUT("IMPOSSIBILE");
               END  IF ; -- fine di IF B...
          ELSE
           X:=-B/A;
           PUT("LA RADICE E'");
           PUT(X);
          END  IF ;--fine di IF A...
```

```
2)  IF  A/=0  THEN
          X:=-B/A;
          PUT("LA RADICE E'");
          PUT(X);
          ELSIF  B=0  THEN
                   PUT("INDETERMINATO");
          ELSE  PUT("IMPOSSIBILE");
         END  IF ;
```

ESERCIZIO 2.10:

```
U:=1.0
WHILE  ABS(U*U-A)/A>0.002  LOOP
    U:=0.5*(U+A/U);
END  LOOP ;
```

ABS(*U-A)/A è la precisione relativa su U2, doppia della precisione su U, da cui lo 0.002 del test.

ESERCIZIO 2.11:

Prima possibilità:

```
K:=0;--oppure N+1
FOR  I  IN  1..N  LOOP
    IF  A(I)=B  THEN
        K:=I; EXIT ;
    END  IF ;
END  LOOP ;
```

L'istruzione EXIT evita di continuare a percorrere la tabella allorché B è trovato, cosa che si farebbe in Pascal (è più conforme alla programmazione strutturata: il ciclo è sempre percorso n volte):

```
K:=0
FOR  I:=1 TO N  DO
     BEGIN
       IF  A(I)=B  THEN  K*=I
     END ;
```

Seconda possibilità:

Ma la soluzione precedente fa tuttavia due test per iterazione e gestisce due variabili K e I. Per evitarlo, si definisce:

AA: ARRAY (NATURAL RANGE < >) OF tipo degli elementi
di A :=A&B;

AA è un tipo anonimo, tabella di dimensione indeterminata di elementi dello stesso tipo di A o B. La dimensione di AA sarà **vincolata** al momento dell'inizializzazione ad A concatenata con B. AA è dunque una matrice di N+1 elementi formata da A e, come ultimo elemento, dalla copia di B.

Ci basta ora scrivere:

```
FOR  K  IN  AA'RANGE  LOOP
     EXIT  WHEN  AA(K)=B;
     END  LOOP ;
```

che si può anche esprimere con un WHILE.

ESERCIZIO 2.12:

```
LOOP
   WHILE  A>B  LOOP  A:=A-B; END  LOOP ;
   WHILE  A<B  LOOP  B:=B-A; END  LOOP ;
EXIT  WHEN  A=B;
END  LOOP ;
```

ESERCIZIO 2.13:

```
DAPAGARE:=IMPORTO;
CASE  LC  IS
   WHEN  'A' 'C'=> DAPAGARE:=0.9*IMPORTO;
   WHEN  'B'    =>  IF  IMPORTO > 1000.0
                     THEN  DAPAGARE:=0.95*IMPORTO;
                     END  IF ;
   WHEN  'D'    => DAPAGARE:=0.95*IMPORTO;

   WHEN  OTHERS  =>  NULL ;
END  CASE ;
```

ESERCIZIO 3.1:

Si possono fare delle conversioni in stringa, dunque:

BB:=B(A(E(FF)));

ESERCIZIO 3.2:

Per DIGITS 1, occorre B>=1/0.3, e B=4 bit.

Le nove mantisse possibili sono: 0,1000 0,1001 0,1010

0,1000	0,1001	0,1010
(1/2)	9/16	5/8

0,1011	0,1100	0,1101	0,1110	0,1111	e 0,0000
11/16	3/4	13/16	7/8	15/16	(0)

Gli esponenti possibili sono compresi fra —16 e +16, quindi una gamma di ordine di grandezze da 2^{-16} ($\sim 10^{-5}$) a 2^{16} ($\sim 10^{5}$).

0.3 vale 3/10; il numero della forma x/16 che è il più vicino è 5/16. Quindi, 0.3 sarà rappresentato da $0,1010*2^{-1}$ che vale 0,3125.

Per DIGITS 3, occorre B=10 bit. Ci sono 513 mantisse possibili (2^9+1 per 0).

Gli esponenti possibili sono compresi fra —40 e +40 dunque una gamma da 2^{-40} (10^{-13}) a 2^{40} (10^{13}).

ESERCIZIO 3.3:

1.0E20 non può essere raggiunto. Per DIGITS 2, occorre B= 7 bit, da cui un esponente massimo di $2^{28} \sim 10^9$.

ESERCIZIO 3.4:

Il delta-binario minimale sarà: 1/27=1/128#0.007

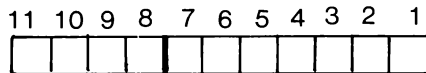
I numeri modello saranno:

numeri voluti:			0	0,01	0,02	0,03		0,04	10,0	
numeri modello:	—15.95....	—0,015	—0,007	0	0,007	0,015	0,022	0,031	0,038	10,0	15,99

Il limite 10 è un numero modello: 10 = 1280 **delta-binario**.

Ma i numeri modello vanno più lontano, occorre n=11 e l'ultimo numero modello è:

$$1/128*(2^{11}-1) = 2047/128 = 15.99.$$

Ciò significa che i numeri saranno compresi in 11 bit, supponendo che la virgola fissa sia fra il bit 7 e il bit 8.

Si vede che i numeri come 0.01 o 0.02 sono rappresentati da una approssimazione. Se si vuole una migliore approssimazione, bisogna specificare un delta più preciso.

ESERCIZIO 3.5:

No, perché in un SUBTYPE, il tipo genitore deve essere esplicitamente designato:

```
SUBTYPE REALE10 IS FLOAT DIGITS 10;
```

Si potrebbe avere:

```
TYPE REALE IS DIGITS 10;
SUBTYPE REALE5 IS REALE DIGITS 5;
```

ESERCIZIO 3.6:

1

ESERCIZIO 3.7:

```
T'POS(T'SUCC(X)) = T'POS(X)+1
```

ESERCIZIO 3.8:

A	B	A/B	A REM B	A MOD B
10	3	3	1	1
9	3	3	0	0
—10	3	—3	—1	2
— 9	3	—3	0	0
10	—3	—3	1	—2
9	—3	—3	0	0
—10	—3	3	—1	—1
— 9	—3	3	0	0

ESERCIZIO 3.9:

```
M'FIRST=1      M'LAST=10      M'LENGTH=10      M'RANGE=1..10
M'FIRST(2)=1   M'LAST(2)=20   M'LENGTH(2)=20   M'RANGE(2)=1..20
```

ESERCIZIO 3.10:

V:=(V'RANGE=> 0.0);

Ciò presuppone che la dimensione di V sia fissata. Altrimenti, un modo di fissare questa dimensione 10 sarebbe:

V:=(1...10 => 0.0);

ESERCIZIO 3.11:

```
TYPE  T  IS ...;
TYPE  TABELLA  IS   ARRAY (1..N) OF  T;
TAB:TABELLA;
TROVATO:BOOLEAN;
K: CONSTANT  T:=...;
```

ESERCIZIO 3.12:

Dichiarazioni dell'esercizio precedente più:

INF, SUP,I:T;

```
INF,SUP,I:T;
BEGIN
  INF:=1;
  SUP:=N;
  TROVATO:=FALSE;
   LOOP
     I:=(INF+SUP)DIV 2;
      IF  TAB(I)=K  THEN  TROVATO:=TRUE;PUT(I);
         ELSIF  TAB(I)<K  THEN  INF:=I+1;
         ELSE  SUP:=I-1;
         END  IF ;
      EXIT  WHEN  TROVATO  OR  INF>SUP;
   END  LOOP ;
   IF  NOT  TROVA  THEN  PUT("NON TROVATO"); END  IF ;
END ;
```

ESERCIZIO 3.13:

```
Y:STRING(1..K):=(1..K => ' ');
IF  K > X'LENGTH  THEN
      Y:=Y(1..K-X'LENGTH)&X;
    ELSIF  K<X'LENGTH  THEN
       Y:=X(1+X'LENGTH-K..X'LENGTH);
    ELSE  Y:=X;
    END  IF ;
```

ESERCIZIO 3.14:

```
 X:STRING;
 Y:STRING;
 L:NATURAL;
BEGIN
   TROVATO:=FALSE;L:=X'LENGTH-1;
    FOR  I  IN  1..Y'LENGTH-L  LOOP
```

```
        IF  Y(I..I+L)=X  THEN
            PUT("TROVATO");PUT(I);
            TROVATO:=TRUE;
             EXIT ;
        END  IF ;
    END  LOOP ;
    IF  NOT  TROVATO  THEN
        PUT("NON TROVATO");
    END  IF ;
END ;
```

ESERCIZIO 3.15:

```
TYPE  LICO  IS
      RECORD
          ETICHETTA:STRING(1..20);
          PREZZOU:FLOAT;
          NUMERO:INTEGER;
          IMPORTO:FLOAT;
      END   RECORD ;
```

Si potrebbe particolarizzare i tipi di PREZZOU e IMPORTO, per esempio:

> SUBTYPE PREZZO IS FLOAT DELTA 0.01 RANGE
> 0.0..10000.0;

e

> PREZZOU:PREZZO;

ESERCIZIO 3.16:

```
IF  IMP.SITFAM='D'  THEN  PUT(EMP.DTVD.ANNO);
                         ELSE  PUT(("NON DIVORZIATO");
END  IF
```

ESERCIZIO 3.17:

Dal punto di vista del funzionamento del programma, nel nostro caso, nessuna differenza. Come scritto nel testo, ogni matrice del tipo CR avrà ovunque per default il valore TRUE. Nella seconda scrittura, soltanto la matrice CRIB è inizializzata a questo valore.

ESERCIZIO 3.18:

```
P:PTR;
P:PARTENZA;
WHILE  P/= NULL    LOOP
     PUT(P.INFO);
     P:=P.SEGUENTE;
END   LOOP ;
```

ESERCIZIO 3.19:

```
  K:STRING;
  P,Q,R:PTR;
BEGIN
  P:=PARTENZA;
   WHILE  P/= NULL   END   THEN  P.INFO<K
        R:=P;
        P:=P.SEGUENTE;
   END   LOOP ;
  Q:= NEW  ELEMENTO ((INFO=>K,SEGUENTE=>P));
  R.SEGUENTE:=Q;
END  INSERZIONE;
```

Notate l'uso di AND THEN nel test. Non bisogna andare a cercare P.INFO allorché P=NULL.

ESERCIZIO 4.1:

Sì, il modo è IN per default.

ESERCIZIO 4.2:

```
PROCEDURE  LISTA  IS
      TYPE  STR10  IS  STRING (1..10);
      TYPE  ELEMENTO;
      TYPE  PTR  IS   ACCESS  ELEMENTO;
      TYPE  ELEMENTO  IS
          RECORD
             INFO: STR10;
             SEGUENTE:PTR;
          END   RECORD
     PAR1,PAR2,PAR3,....:PTR;-- puntatori di partenza di un
                             -- certo numero di liste
                             -- analoghe.


     X:STR10;

      FUNCTION  PRESENTE (K: IN  STR10;PAR: IN  PTR)
                  RETURN  BOOLEANO  IS  P:PTR;


      BEGIN
        P:=PAR;
         WHILE  P/= NULL   END   THEN  P.INFO/=K  LOOP
              P:=P. SEGUENTE;
         END   LOOP ;
         IF  P= NULL   THEN   RETURN  FALSE;
                  ELSE   RETURN  TRUE;
         END   IF ;
    END  PRESENTE;



    FUNCTION  POSIZIONE (K: IN  STR10;PAR: IN PTR)
                RETURN PTR  IS  P:PTR;
      BEGIN
        P:=PAR;
         WHILE  P/= NULL  END   THEN  P.INFO/=K LOOP
              P:=P.SEGUENTE;


         END   LOOP ;
         RETURN  P;  END  POSIZIONE;
       BEGIN --LISTA
      -- costruzione delle liste puntate da PAR1, PAR2, PAR3...
      GET(X);-- lettura di un elemento da cercare
       IF  PRESENTE(K=>X,PAR=>PAR3)...
      PUT(PRESENTE(X,PAR2));
       IF  POSIZIONE(PAR=>PAR2,K=>K)/= NULL ....
      PAR3:=POSIZIONE(X,PAR3).SEGUENTE;
      .
      .
      .
END  LISTA;
```

L'ultima istruzione PAR3 := ha per effetto di sopprimere dalla lista puntata da PAR3 tutti gli elementi fino a, e compreso quello, la cui informazione è uguale a X:

ESERCIZIO 4.3:

```
PROCEDURE  LISTA  IS
      TYPE  STR10  IS  STRING(1...10);
      TYPE  ELEMENTO;
      TYPE  PTR  IS   ACCESS  ELEMENTO;
      TYPE  ELEMENTO  IS
          RECORD
             INFO:STR10;
             SEGUENTE:PTR;
          END   RECORD ;
     PAR1, PAR2, PAR3...: PTR;
     X:STR10;

      PROCEDURE  INSERZIONE(K:  IN  STR 10;PAR: IN  PTR);
         P,Q,R:PTR;
        BEGIN
         P:=PAR;
            WHILE  P/= NULL   END   THEN  P.INFO<K LOOP
              R:=P;
              P:=P. SEGUENTE;
            END   LOOP ;
          Q:= NEW  ELEMENTO(K,P);
          R.SEGUENTE:=Q;
         END  INSERZIONE;

       BEGIN -- LISTA
      -- creazione delle liste
         GET(X);
         INSERZIONE (X,PAR2);
         INSERZIONE(K=>"TAGADAGADA",PAR=>PAR1);
       END  LISTA
```

Note:

1) Questa procedura INSERZIONE è diversa da quella del capitolo 1 che consisteva nell'aggiungere un elemento in cima alla lista puntata da INTO.

2) I riferimenti a ELEMENTO, a SUCCESSIVO, ecc.. nella procedura sono legali, poiché questi dati non essendo stati ridefiniti nella procedura, sono globali.

ESERCIZIO 4.4:

```
TYPE   MATRICE   IS    ARRAY   (INTEGER   RANGE  <>, INTEGER
RANGE  <>)  OF  FLOAT;
PROCEDURE  SCAMBIO(M1,M2: IN   OUT  MATRICE)  IS
     X:MATRICE;
   BEGIN
     X:=M1;
     M1:=M2;
     M2:=X;
   END  SCAMBIO;
```

ESERCIZIO 4.5:

```
PROCEDURE  PRODMA(A,B: IN  MATRICE,C  OUT  MATRICE)  IS

      Z:FLOAT;
       BEGIN
              IF      A'RANGE(2)/=B'RANGE      THEN        RAISE
ERRORE-LUNGHEZZA;
            END   IF ;
            FOR  I  IN  A'RANGE  LOOP
               FOR  J  IN  B'RANGE(2)  LOOP
                  Z:=0.0;
                   FOR  K  IN  B( RANGE  LOOP
                       Z:=Z.A(I,K)*B(K,J);
                   END   LOOP ;
                  C(I,J):=Z;
               END   LOOP ;
           END   LOOP ;
       END  PRODMA;
```

ESERCIZIO 4.8:

```
PROCEDURE  HANOI  IS
    N:INTEGER;
     PROCEDURE  MVT(I,J: IN  INTEGER);
         BEGIN
            PUT(I);PUT("=>");PUT(J);
         END  MVT;
     PROCEDURE  SPOS (N,I,J: IN  INTEGER);
         BEGIN
             IF  N=1  THEN  MVT(I,J);
                    ELSE
                      SPOS(N-1,I,6-I-J);
                      SPOS(1,I,J);
                      SPOS(N-1,6-I-J,J);
             END  IF ;
          END  SPOS;
     BEGIN
         GET (N);
         SPOS(N,1,2);
     END  HANOI;
```

ESERCIZIO 4.9:

No, non si ha diritto all'assegnazione all'esterno del package per un tipo LIMITED PRIVATE. Si sarebbe avuto il diritto se il tipo fosse stato soltanto PRIVATE.

Si avrebbe tuttavia il diritto di scrivere:

R:RISORSA;
GETRISORSA(......,R);
ACCESSORISORSA(.......,R);

ESERCIZIO 4.10:

```
PACKAGE  GESTIONE-PILA  IS
    TYPE  T  IS ...;-- potrebbe anche essere in un dominio
                    -- che circoscrive la definizione del package.
    PROCEDURE  AGGIUNGERE (OGGETTO:T);
    FUNCTION  TOGLIERE  RETURN  T;
   PILA-PIENA,PILA-VUOTA:BOOLEANO;
END  GESTIONE-PILA;
```

Per una gestione mediante matrice, il corpo si scriverà:

```
PACKAGE  BODY  GESTIONE-PILA  IS
    MISURA:  CONSTANT  INTEGER:=100;-- o ogni altro valore
                                   -- deciso dal programmatore
    PILA: ARRAY (1..MISURA)  OF  T;
    PUNTATORE:INTEGER  RANGE  0..MISURA;
     PROCEDURE  AGGIUNGERE (OGGETTO:T)  IS
        BEGIN
          PUNTATORE:=PUNTATORE + 1;
          PILA(PUNTATORE):=OGGETTO;
           IF  PUNTATORE=MISURA  THEN  PILA-PIENA:=TRUE;
           END  IF ;
        END  AGGIUNGERE;

     FUNCTION  TOGLIERE  RETURN  T  IS
        BEGIN
          PUNTATORE:=PUNTATORE-1;
           IF  PUNTATORE<=0  THEN  PILA-VUOTA:=TRUE;
           END  IF ;
           RETURN  PILA (PUNTATORE+1);
        END  TOGLIERE;

     BEGIN  -- inizializzazioni
       PUNTATORE:=0;
       PILA-VUOTA:=TRUE;
       PILA-PIENA:=FALSE;
     END  GESTIONE-PILA;
```

Per una gestione in memoria dinamica, si avrebbe:

```
PACKAGE  BODY  GESTIONE-PILA  IS  -- versione 2
    MISURA: CONSTANT :=100;
     TYPE  ELEM;
     TYPE  PTR  IS  ACCESS  ELEM;
     TYPE  ELEM  IS
         RECORD
          DATO:T;
          PREC:PTR;
         END  RECORD ;
    PUNTATORE:PTR;
    NUMERO:INTEGER  RANGE 0..MISURA;
     PROCEDURE  AGGIUNGERE(OGGETTO,T)  IS
        P:PTR;
         BEGIN
             NUMERO:=NUMERO+1;
              IF  NUMERO=MISURA  THEN  PILA-PIENA;=TRUE;
              END  IF ;
             P:= NEW  ELEM((OGGETTO,PUNTATORE));
             PUNTATORE:=P;
         END  AGGIUNGERE;
     FUNCTION  TOGLIERE  RETURN  T  IS
        X:ELEM;
         BEGIN
             X:=PUNTATORE.  ALL ;
```

```
-- si potrebbe aggiungere una liberazione esplicita di
-- area di memoria
      PUNTATORE:=X PREC;
      NUMERO:=NUMERO-1;
       IF  NUMERO<=0  THEN  PILA-VUOTA:=TRUE;
       END  IF ;
       RETURN  X.DATO;
  END  TOGLIERE;

BEGIN  -- inizializzazioni
   PUNTATORE:= NULL ;
   NUMERO:=0;
   PILA-VUOTA:=TRUE;
   PILA-PIENA:=FALSE;
END  GESTIONE-PILA;
```

una utilizzazione corretta sarà della forma:

```
USE  GESTIONE PILA:
X,Y:T;        nella parte dichiarazioni.
IF   NOT  PILA-VUOTA  THEN  X:=TOGLIERE ();
END   IF ;
IF   NOT  PILA-PIENA  THEN  AGGIUNGERE (X);
END   IF ;
```

Delle chiamate non protette con un test come qui sopra possono causare delle eccezioni. Ora, i test potrebbero essere incorporati nelle procedure. Notare la chiamata della funzione con la parentesi vuota, poiché non vi sono argomenti.

ESERCIZIO 4.11:

Si avrà:

```
PROCEDURE  UT-PILA  IS
      PACKAGE  GESTIONE-PILA  IS
     -- la specificazione senza cambiamento.
      END  GESTIONE-PILA;
      PACKAGE   BODY  GESTIONE-PILA  IS   SEPARATE ;
BEGIN  -- UT-PILA
   .
   .
END  UT-PILA;

Una seconda compilazione conterra':

SEPARATE (UT-PILA)
PACKAGE   BODY  GESTIONE-PILA  IS
```

```
  -- dichiarazioni, poi
   PROCEDURE  AGGIUNGERE (OGGETTO:T) IS  SEPARATE ;
   FUNCTION  TOGLIERE  RETURN  T  IS  SEPARATE ;
END  GESTIONE-PILA;

La terza compilazione comprendera':

SEPARATE  (UT-PILA.GESTIONE-PILA)
PROCEDURE  AGGIUNGERE-OGGETTO:T)  IS
END  AGGIUNGERE;
```

Stessa cosa per TOGLIERE.

ESERCIZIO 4.12:

Basterà scrivere (nell'ipotesi della gestione mediante tabella),

```
GENERIC
   MISURA:INTEGER;
    TYPE  T  IS  PRIVATE ;
PACKAGE  GESTIONE-PILA  IS
    PROCEDURE  AGGIUNGERE....(senza cambiamento)
END  GESTIONE-PILA;
PACKAGE  BODY  GESTIONE-PILA  IS
   PILA: ARRAY (1..MISURA)  OF  T;
   .
   .
   .
   .
END  GESTIONE-PILA;
```

Si potrà istanziarlo per esempio per una catasta di 200 interi ed una di 100 reali:

```
PACKAGE PILAINT IS NEW GESTIONE-PILA(200,INTEGER);
PACKAGE PILA-REALI IS NEW GESTIONE-PILA(DIMENSIONE=>
100,T => FLOAT);
```

Si potrà, per esempio, scrivere:

```
I:INTEGER;
R:FLOAT;
```

```
I:=PILAINT.TOGLIERE( );
PILA-REALI.AGGIUNGERE(R);
```

Si sarebbe potuto anche scrivere:

```
PROCEDURE IMP-R(R:FLOAT) RENAMES
PILA-REALI.AGGIUNGERE;
```

poi

```
IMP-R(R);
```

ESERCIZIO 4.13:

```
FUNCTION  F(X:FLOAT) RETURN  FLOAT  IS
    -- definizione della funzione F
END  F;
FUNCTION   INTEG(A,B:   IN   FLOAT;N:  IN   INTEGER)  RETURN
FLOAT  IS
    S,HX:FLOAT;
BEGIN
     S:=(F(A)+F(B))/2.0;
     H:=(B-A)/N;
     X:=A;
      FOR  I  IN  1...N-1  LOOP
        X:=X+H;
        S:=S+F(X);
      END  LOOP ;
      RETURN  S;
END  INTEG;
```

ESERCIZIO 4.14:

```
GENERIC
    WITH   FUNCTION  F (X:FLOAT)  RETURN  FLOAT;
FUNCTION   INTEG  (A,B:  IN   FLOAT; N: IN  INTEGER)  RETURN
FLOAT  IS
   -- il resto senza cambiamento.
    FUNCTION  INTEG-SEN  IS   NEW  INTEG (SEN);
    FUNCTION  INTEG-COS  IS   NEW  INTEG (COS);
```

Le chiamate potranno essere della forma:

```
Z:INTEG-SEN (0.0,2.0*PI,N=>50);
W:=INTEG-COS(0.0,PI/2.0,20);
```

ESERCIZIO 5.1:

```
PROCEDURE  AGGIUNGI  IS
    PACKAGE  E-S-REALI  IS   NEW  INPUT-OUTPUT (FLOAT);
    USE  E-S-REALI;
   X:FLOAT;
   SCHEDARIO: INOUT-FILE;
BEGIN
   OPEN (SCHEDARIO,"XXXX");
   SET-WRITE(SCHEDARIO,LAST(SCHEDARIO)+1);
   -- calcolo di X
   WRITE 'SCHEDARIO,X);
   CLOSE(SCHEDARIO);
END  AGGIUNGI;
```

ESERCIZIO 5.2

```
PROCEDURE  AGGIORNAMENTO  IS
      TYPE  IMPIEGATO  IS
           RECORD
            NUMERO:INTEGER;
            NOME:STRING(1..10);
            INDIRIZZO:STRING(1..30);
            INSEE:LONG-INTEGER;
            IMPIEGO:STRING(1..10);
            GRADO:INTEGER;
           END   RECORD ;
      PACKAGE  ES  IS   NEW  INPUT-OUTPUT(IMPIEGATO);
      USE  ES;
     SCHEDPERS:INOUT-FILE;
     MOV:IN-FILE;
     IMP:IMPIEGATO;
     N:FILE-INDICE;
```

```
BEGIN
  OPEN (SCHEDPERS,-XXXX);
  OPEN (MOV,"YYYY");
   WHILE   NOT   END - OF -FILE (MOV)   LOOP
        READ (MOV,IMP);
        N:= FILE-INDEX (IMP.NUMERO);
        SET-WRITE(SCHED-PERS,TO=>N);
        WRITE( SCHED-PERS,IMP);
    END   LOOP ;
   CLOSE(SCHEDPERS);
   CLOSE(MOV);
END  AGGIORNAMENTO;
```

ESERCIZIO 5.3:

```
PROCEDURE  COPIA  IS
       USE  TEXT-IO;
      ZOZO:IN-FILE;
      ZAZA:OUT-FILE;
      X:STRING;
BEGIN
      CREATE(ZAZA,"XXXX");
      OPEN(ZOZO,"ZZZZ");
      SET INPUT(ZOZO);
      SET OUTPUT(ZAZA);
       WHILE   NOT   END - OF -FILE(ZOZO) LOOP
            X:=GET-LINE();
            PUT-LINE(X);
       END   LOOP ;
      CLOSE(ZOZO); CLOSE(ZAZA);
END   COPIA;
```

ESERCIZIO 5.4:

'150'
'□□□□□□—150'
'□□□□2#10010110#'

ESERCIZIO 5.5:

```
PROCEDURE  SALUTO  IS
     USE  TEXT-IO;
     TYPE  GIORNO  IS  (LUNEDI,MARTEDI,MERCOLEDI,GIOVEDI,VENERDI,
SABATO,DOMENICA);
    TODAY:GIORNO;
     PACKAGE  E-S  IS  NEW  ENUMERAZIONE-IO(GIORNO);  USE  E-S;
BEGIN
    PUT("CHE GIORNO E' ?");
    GET(TODAY);
     NEW -LINE(SPACING=>3);
    PUT("****");
     IF  GIORNO  IN  SABATO....DOMENICA  THEN
                  PUT("BUON WEEKEND");
                                          ELSE
                  PUT("BUONGIORNO");
      END  IF ;
     PUT("****");
END  SALUTO;
```

Esempio di dialogo (ciò che è battuto dall'utente è sottolineato):

CHE GIORNO È? <u>SABATO</u>

___ __ ____

__ __ ______

****BUON WEEK END ****

ESERCIZIO 6.1:

```
PROCEDURE  ZOZO  IS
     X:type;
      TASK  T  IS
         ENTRY  PRONTO;
      END ;
      TASK  BODY  T  IS
      BEGIN
          .
          . istruzioni 1
          .
          ACCEPT  PRONTO;
          .
```

```
     . istruzioni 3
     .
   END  T;
   TASK  U  IS
   END ;
   TASK  BODY  U  IS
   BEGIN
     .
     . istruzioni 2
     .
     T.PRONTO;
   END  U;
BEGIN --ZOZO e' ora vuoto
   NULL ;
   END  ZOZO;
```

Si approfitta ora della simultaneità possibile fra 1 e 2. Su di un monoprocessore, il tempo passato sarà lo stesso che con la prima versione, ma si ha un guadagno potenziale se si dispone di un multiprocessore.

ESERCIZIO 6.2:

Uno solo, poiché, alla prima chiamata in attesa il compito U è sospeso, non può dunque fare una seconda chiamata. D'altronde, nell'insieme delle liste d'attesa, non può esservi che zero o una chiamata proveniente da U.

ESERCIZIO 6.3:

```
TASK  SEM  IS
   ENTRY  P;
   ENTRY  V;
END ;
TASK  BODY  SEM  IS
BEGIN
   LOOP
     ACCEPT  P;
     ACCEPT  V;
   END  LOOP ;
END  SEM;
```

Sì, ed è altrettanto semplice! Dal momento in cui P è stato accettato, non può essere fatto nessun altro accesso alla risorsa poiché il semaforo accetta una sola entrata V. Allorché il task chiamante ha eseguito SEM.V, ed ha liberato la risorsa, si è pronti ad accettare (grazie al ciclo) la prossima richiesta P.

Infatti, i semafori sono un mezzo di gestione della simultaneità più primitivi dei meccanismi offerti da ADA, non fa dunque meraviglia che la traduzione in ADA sia molto semplice.

ESERCIZIO 6.4:

Basta aggiungere:

OR TERMINATE;

prima di

END SELECT;

ESERCIZIO 6.5:

```
--dichiarazioni globali:
TYPE  T LINEA  IS  STRING(1...132);
--nel task produzione:
LOOP
   .
   .
LINEA:=....;
SPOOLING.SCRITTO (LINEA);--rallenta se l'entrata SCRITTO
   .                     --non e' accettata
   .
   .
END  LOOP ;
-- nel task stampa:
LOOP
   .
   .
   .
```

```
SPOOLING. LEGGE(LINEA);-- se non c'e' una linea da stampare nel
PUT(LINEA);              -- buffer, l'entrata non e' accettata,
END   LOOP ;
   TASK  SPOOLING  IS
        ENTRY  LEGGE(LINEA:  OUT  TLINEA);
        ENTRY  SCRIVE(LINEA:IN TLINEA);
END ;
TASK   BODY  SPOOLING  IS
       MISURA:  CONSTANT  INTEGER:=2000;-- valore deciso dal
                                        -- sistema
       BUFFER: ARRAY (1..MISURA)  OF  TLINEA;
       PRIMO-LIBERO,PROSSIMO-OCCUPATO:INTEGER  RANGE  1..MISURA :=1;
       NUMERO: INTEGER  RANGE  0..MISURA:=0;-- numero di linee
                                            -- nel buffer.

BEGIN
    LOOP
       SELECT
           WHEN  NUMERO<MISURA=>--accetta nuova linea se non pieno
               ACCEPT  SCRIVE (LINEA: IN  TLINEA)  DO
                  BUFFER(PRIMO-LIBERO):= LINEA;
               END ;-- DO
              PRIMO-LIBERO:=PRIMO-LIBERO  MOD  MISURA+1;
              NUMERO:=NUMERO+1;
        OR
           WHEN  NUMERO MAGGIORE DI 0=>-- puo' fornire una linea
                                      -- finche' ce n'e'


           ACCEPT  LEGGE(LINEA: OUT  TLINEA)  DO
               LINEA:=BUFFER(PROSSIMO-OCCUPATO);
                END ;-- DO
               PROSSIMO-OCCUPATO:=PROSSIMO-OCCUPATO  MOD  MISURA+1;
         OR
           TERMINATE ;
         END  SELECT ;
       END   LOOP ;
   END  SPOOLING;
```

ESERCIZIO 6.6:

```
   .
   .
   .
LINEA: =...
SELECT
    SPOOLING. SCRIVE (LINEA);
OR
     DELAY  0.05; PUT(CONSOLE,"SOVRACCARICO STAMPANTE");
END   SELECT ;
   .
   .
   .
```

ESERCIZIO 7.1:

Si aggiungono le dichiarazioni seguenti nel corpo del package (dopo PUNTATORE:...).

TROPPO,VUOTA: EXCEPTION

poi, in AGGIUNGERE:

```
BEGIN
    IF  PILA-PIENA  THEN   RAISE  TROPPO; END   IF ;
   PUNTATORE:=...
    .
    .
    .
    END   IF ;
EXCEPTION
    WHEN  TROPPO=>
       PUT("PILA PIENA");
END  AGGIUNGERE;


in TOGLIERE :


BEGIN
    IF  PILA-VUOTA  THEN   RAISE  VUOTA; END   IF ;
    .
    .
    .
EXCEPTION
     WHEN  VUOTA =>
        PUT("PILA VUOTA");
         RETURN  PILA (1);--non obbligatoria.
END  TOGLIERE;
```

Si potevano anche mettere le routine di trattamento nel programma chiamante.

ESERCIZIO 7.2:

Occorre la dichiarazione:

```
ERRORE-LUNGHEZZA: EXCEPTION ;

prima di  BEGIN

  Prima di  END  PRODMA; si puo' mettere :
EXCEPTION
     WHEN  ERRORE-LUNGHEZZA=>
        PUT("DIMENSIONI NON CONFORMI");  NEW -LINE;
        PUT("MATRICE C MESSA A ZERO");
        C:=(A'RANGE =>(B  RANGE (2)=>0.0));
END  PRODMA;
```

ESERCIZIO 8.1:

Ci sono circa 20.000 numeri modello mentre 12 bit non possono codificare che 4096 valori diversi. Bisogna avere almeno:

FOR REALE'SIZE USE 15;

ESERCIZIO 8.3:

```
BYTE*= CONSTANT =1;
TYPE  STATUS-REGISTER COPY  IS
     RECORD
     C:BOOLEANO;
     Z:BOOLEANO;
     I:BOOLEANO;
     D:BOOLEANO;
     B:BOOLEANO;
     V:BOOLEANO;
     N:BOOLEANO;
     END   RECORD
```

```
FOR   STATUS-REGISTER-COPY   USE
       RECORD
       C  AT   0*BYTE   RANGE  0..0;
       Z  AT   0*BYTE   RANGE  1..1;
       I  AT   0*BYTE   RANGE  2..2;
       D  AT   0*BYTE   RANGE  3..3;
       B  AT   0*BYTE   RANGE  4..4;
       V  AT   0*BYTE   RANGE  6..6;
       N  AT   0*BYTE   RANGE  7..7;
       END   RECORD ;
```

ESERCIZIO 8.5:

```
PIA1,PIA2:PIA;
FOR PIA1 USE AT 16#E810#;
FOR PIA2 USE AT 16#E820#,
```

ESERCIZIO 8.6:

È sufficiente faré:

```
TYPE V8 IS RANGE 0...255;
VARIABLE:V8;
FOR V8'SIZE USE 8;
FOR VARIABLE USE AT INDIRIZZO VOLUTO;
```

ed è sufficiente l'assegnazione: VARIABLE :=valore voluto;

APPENDICE 5

BIBLIOGRAFIA

Dal 1979, le riviste (in inglese) "ACM SIGPLAN Notices" e "Software Practice and Experience" pubblicano numerosi articoli su ADA in ciascun numero.

Questi articoli sono sia delle valorizzazioni, sia delle discussioni del modo in cui si può realizzare un modulo particolare di un compilatore.

Citiamo in particolare:

D.V. MOFFATT - Enumerations in Pascal, Ada and beyond
ACM Sigplan Notices 16,2,77,1981

J.D. ICHBIAH et al. Preliminary Ada Reference Manual
ACM Sigplan Notices 14,6-a 1979

J.D. ICHBIAH et al. Rationale for the Design of the
Ada Programming Language
ACM Sigplan Notices 14,6-b 1979

Il numero di novembre 1980 costituisce i Proceedings of the ACM-SIGPLAN symposium on the Ada Programming Language-ACM Sigplan Notices 15,11. 1980.

Il manuale di riferimento definitivo (Reference manual for the Ada programming language) è disponibile presso il Cie CII HB (Louveciennes) o presso il Dipartimento della Difesa U.S.A..

Il Dipartimento della Difesa pubblica anche i rapporti successivi che hanno formato e affinato il quaderno dei progetti, i più recenti essendo :"Ironman", "Steelman" e "Stoneman" (quest'ultimo definisce gli imperativi dell'ambiente della programmazione (APSE)).

Si potrà consultare il documento associato al "Tutorial on Ada" di J. Barnes, pubblicato dall'Association Euromicro o i documenti pubblicati dalla società ALSYS.

Dal punto di vista dei libri, noi non conosciamo all'autunno del 1985 che:

P. WEGNER — Programming with ADA
Prentice Hall
(che tratta di ADA preliminare) 1980

e:

D. Bjorner, O.N. Oest Ed— Towards a formal description of Ada — springer 1980
(che si rivolge piuttosto ai futuri realizzatori di un compilatore ADA).

Nel 1975 il Dipartimento della Difesa degli Stati Uniti decise di bandire un concorso per un linguaggio che riassumesse in sé le caratteristiche positive dei linguaggi allora disponibili e che consentisse una drastica riduzione dei costi di produzione e manutenzione del software.
Il concorso fu vinto dalla Honeywell-Bull con la prima versione di ADA, che prende il nome dalla Contessa Ada Augusta Lovelace, figlia del poeta Byron e collaboratrice di Charles Babbage, inventore del primo computer della storia, mai realizzato in pratica.
Il nuovo linguaggio è stato creato per soddisfare stringenti requisiti di portabilità, potenza, flessibilità e facilità di manutenzione, superando così le carenze classiche dei più comuni linguaggi procedurali.

L. 19.500

COD. 540A - ISBN - 88-7056-206-9